James H. Davenport
William M. Farmer
Florian Rabe
Josef Urban (Eds.)

LNAI 6824

Intelligent Computer Mathematics

18th Symposium, Calculemus 2011
and 10th International Conference, MKM 2011
Bertinoro, Italy, July 2011, Proceedings

Lecture Notes in Artificial Intelligence 6824

Edited by R. Goebel, J. Siekmann, and W. Wahlster

Subseries of Lecture Notes in Computer Science

James H. Davenport William M. Farmer
Florian Rabe Josef Urban (Eds.)

Intelligent Computer Mathematics

18th Symposium, Calculemus 2011
and 10th International Conference, MKM 2011
Bertinoro, Italy, July 18-23, 2011
Proceedings

Springer

Volume Editors

James H. Davenport
University of Bath, Departments of Computer Science and Mathematics Sciences
Bath BA2 7AY, UK
E-mail: j.h.davenport@bath.ac.uk

William M. Farmer
McMaster University, Department of Computing and Software
1280 Main Street West, Hamilton, ON L8S 4K1, Canada
E-mail: wmfarmer@mcmaster.ca

Florian Rabe
Jacobs University Bremen, Computer Science
Campus Ring 1, 28759 Bremen, Germany
E-mail: f.rabe@jacobs-university.de

Josef Urban
Radboud University, Institute for Computing and Information Sciences
Heyendaalseweg 135, 6525 AJ Nijmegen, The Netherlands
E-mail: josef.urban@gmail.com

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-22672-4 e-ISBN 978-3-642-22673-1
DOI 10.1007/978-3-642-22673-1
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2011932178

CR Subject Classification (1998): I.2, H.3, H.4, F.4.1, H.2.8, F.1

LNCS Sublibrary: SL 7 – Artificial Intelligence

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

As computers and communications technology advance, greater opportunities arise for intelligent mathematical computation. While computer algebra, automated deduction, mathematical databases/tables and mathematical publishing each have long and successful histories, we are now seeing increasing opportunities for synergy among them. The Conferences on Intelligent Computer Mathematics series hosts collections of co-located meetings, allowing researchers and practitioners active in these related areas to share recent results and identify the next challenges.

2011 marked the fourth in this series of Conferences on Intelligent Computer Mathematics (CICM), held in Italy (Bertinoro). Previous conferences, all also published in Springer's *Lecture Notes in Artificial Intelligence* series, have been held in the UK (Birmingham, 2008: LNAI 5144), Canada (Grand Bend, Ontario, 2009: LNAI 5625) and France (Paris, 2010: LNAI 6167). CICM 2011 included two long-standing international conferences:

- 18th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning (Calculemus 2011)
- 10th International Conference on Mathematical Knowledge Management (MKM 2011)

The more coordinated evolution of CICM has made it possible to add a new track for brief descriptions of systems and projects that span the Calculemus and MKM topics (the "S&P" track), along with the Calculemus and Mathematical Knowledge Management (MKM) tracks. The proceedings of these three tracks are collected in this volume. The reader comparing this volume with its predecessors should note that Artificial Intelligence and Symbolic Computation (AISC) is only held in even years, and hence is not present here.

CICM 2011 also contained the following activities:

- 4th Workshop Towards a Digital Mathematics Library (DML 2011)
- Demonstrations of the systems presented in the S&P track
- Less formal "work in progress" sessions

For 2011, we used the "multi-track" features of the EasyChair system, and our thanks are due to Andrei Voronkov and his team for this and many other features. The multi-track feature also allowed transparent handling of conflicts of interest between the Track Chairs and submissions: these submissions were moved to a separate track overseen by the General Chair. The EasyChair conference statistics are the following.

> There were 51 submissions. Each submission was reviewed by at least 2, and on average 4.1, Programme Committee members. The committee decided to accept 30 papers.

However, this is a conflation of tracks with different acceptance characteristics. The track-based acceptance rates were:

Calculemus—9 acceptances out of 15 submissions
MKM—9 acceptances out of 22 submissions
Systems and Projects—12 acceptances out of 14 submissions

May 2011

James Davenport
William Farmer
Florian Rabe
Josef Urban

Organization

CICM 2011 was organized by the Conferences in Intelligent Computer Mathematics special interest group, which was formed at CICM 2010 as a parent body to the long-standing Calculemus and Mathematical Knowledge Management special interest groups. The corresponding conferences organized by these interest groups continue as special tracks in the CICM conference. These tracks and the new Systems and Projects track had independent Track Chairs and Programme Committees. The newly added Systems and Projects track publishes abstracts about systems and projects related to MKM and Calculemus topics, and about progress on existing systems and projects. The system demonstrations and project posters were presented in dedicated CICM sessions.

Local arrangements, the life-blood of any conference, were handled by the Department of Computer Science of the Università di Bologna, Italy.

CICM Steering Committee

Secretary	Michael Kohlhase	Jacobs University Bremen, Germany
MKM/Treasurer	William Farmer	McMaster University, Canada
Calculemus	Renaud Rioboo	ENSIIE, France

Calculemus Trustees

Stephen Watt
Paul Jackson
Thierry Coquand
Marc Moreno Maza
James Davenport
Makarius Wenzel
Renaud Rioboo
David Delahaye
Lucas Dixon
Jacques Carette
William Farmer

Mathematical Knowledge Management Trustees

Petr Sojka
James Davenport
Serge Autexier
William Farmer
Claudio Sacerdoti Coen
Alan Sexton
Florian Rabe

CICM 2011 Officers

General Chair	James Davenport	University of Bath, UK
Local Arrangements	Andrea Asperti	Università di Bologna, Italy
Calculemus Track Chair	William Farmer	McMaster University, Canada
MKM Track Chair	Florian Rabe	Jacobs University, Germany
S&P Track Chair	Josef Urban	Radboud University Nijmegen, The Netherlands

Programme Committee for the Calculemus Track

William Farmer (Chair)	McMaster University, Canada
Thorsten Altenkirch	University of Nottingham, UK
Serge Autexier	DFKI, Germany
Christoph Benzmüller	Articulate Software, Angwin, CA, USA
Anna Bigatti	Università di Genova, Italy
Herman Geuvers	Radboud University Nijmegen, The Netherlands
Cezary Kaliszyk	University of Tsukuba, Japan
Deepak Kapur	University of New Mexico, USA
Assia Mahboubi	INRIA, France
Francisco-Jesús Martín-Mateos	Universidad de Sevilla, Spain
Russell O'Connor	McMaster University, Canada
Grant Passmore	University of Cambridge, UK
Silvio Ranise	Fondazione Bruno Kessler, Italy
Alan Sexton	University of Birmingham, UK
Adam Strzeboński	Wolfram Research Inc., USA

Programme Committee for the Mathematical Knowledge Management Track

Florian Rabe (Chair)	Jacobs University Bremen, Germany
Laurent Bernardin	Maplesoft, Canada
Thierry Bouche	Université Joseph Fourier, Grenoble, France
Simon Colton	Imperial College, London, UK
Patrick Ion	American Mathematical Society
Johan Jeuring	University of Utrecht, The Netherlands
Fairouz Kamareddine	Heriot–Watt University, Edinburgh, UK
Manfred Kerber	University of Birmingham, UK
Andrea Kohlhase	DFKI, Germany
Paul Libbrecht	University of Education, Karlsruhe, Germany
Bruce Miller	NIST, USA
Adam Naumowicz	Uniwersytet w Białymstoku, Poland
Claudio Sacerdoti Coen	Università di Bologna, Italy
Petr Sojka	Masaryk University, Brno, Czech Republic
Volker Sorge	University of Birmingham, UK
Masakazu Suzuki	Kyushu University, Japan
Enrico Tassi	Microsoft Research–INRIA Joint Centre, France
Makarius Wenzel	Université Paris-Sud, France
Freek Wiedijk	Radboud University Nijmegen, The Netherlands

Programme Committee for the Systems and Projects Track

Josef Urban (Chair)	Radboud University Nijmegen, The Netherlands
Andrea Asperti	Università di Bologna, Italy
Michael Beeson	San José State University, CA, USA
Jacques Carette	McMaster University, Canada
Michael Kohlhase	Jacobs University Bremen, Germany
Christoph Lange	Jacobs University Bremen, Germany
Piotr Rudnicki	University of Alberta, Canada

Additional Referees

In addition to the many members of the Programme Committees who refereed for other tracks, the Programme Chairs are grateful to the following.

Baker, Josef	Huisman, Marieke	Strecker, Martin
Coglio, Alessandro	Mamane, Lionel	Sturm, Thomas
Dietrich, Dominik	Mayero, Micaela	Tankink, Carst
Ferreira, João	McCune, William	
Harrison, John	Spitters, Bas	

Sponsoring Institutions

Università di Bologna.

Table of Contents

Calculemus 2011

Mathematical Knowledge Management 2011

CICM Systems and Projects

Enumeration of AG-Groupoids

Andreas Distler[1,⋆], Muhammad Shah[2,⋆⋆], and Volker Sorge[3]

[1] Centro de Álgebra,
Universidade de Lisboa,
Lisboa, Portugal
andreas@mcs.st-andrews.ac.uk
[2] Department of Mathematics,
Quaid-i-Azam University,
Islamabad, Pakistan
shahmaths_problem@hotmail.com
[3] School of Computer Science,
The University of Birmingham
Birmingham, UK
V.Sorge@cs.bham.ac.uk

Abstract. Enumeration and classification of mathematical entities is an important part of mathematical research in particular in finite algebra. For algebraic structures that are more general than groups this task is often only feasible by use of computers due to the sheer number of structures that have to be considered. In this paper we present the enumeration and partial classification of AG-groupoids — groupoids in which the identity $(ab)c = (cb)a$ holds — of up to order 6. The results are obtained with the help of the computer algebra system GAP and the constraint solver Minion by making use of both algebraic techniques as well as search pruning via symmetry breaking.

1 Introduction

The classification of mathematical structures is an important branch of research in pure mathematics. In particular, in abstract algebra the classification of algebraic structures is an important pre-requisite for their goal-directed construction to make them amenable in practical applications. For example, the classification of finite simple groups — which was described as one of the major intellectual achievements of the twentieth century [11] — not only allows to immediately compute the number of non-isomorphic, simple groups of a particular finite order but also gives a concrete recipe how to construct a representant for each class.

While full classification of structures is usually the goal, an important first step towards this goal is often the enumeration of structures with particular properties. Enumeration results can be obtained by a variety of means, depending on

⋆ The work was supported at CAUL within the projects PTDC/MAT/101993/2008 and ISFL-1-143, financed by FEDER and FCT.

⋆⋆ The work was done while at the School of Computer Science, University of Birmingham, with financial support by the Ramsay Research Fund.

J.H. Davenport et al. (Eds.): Calculemus/MKM 2011, LNAI 6824, pp. 1–14, 2011.

the domain, for example by combinatorial or algebraic considerations. However, in algebraic domains where the objects under consideration exhibit little in way of internal structure, exhaustive generation is often the most reliable means of obtaining useful enumeration results. As a consequence, a number of projects have been concerned with automatic enumeration of algebraic structures that are more general than groups.

Quasigroups and loops — two types of non-associative structures with Latin square property — have been enumerated up to size 11 using a mixture of combinatorial considerations and bespoke exhaustive generation software [17,16]. In other approaches general purpose automated reasoning technology has been employed. For instance the model generator Finder (Finite Domain Enumerator) [32] has been used for obtaining novel enumeration results, most recently for IP-loops up to size 13 [1], but also has been successfully employed to solve open questions in quasigroup theory. Going beyond pure enumeration is the generation of classification theorems that provide discriminating algebraic properties for different isomorphism and isotopism classes of quasigroups and loops up to size 7 using a combination of theorem proving, model generation, satisfiability solving and computer algebra [33]. For the case of associative structures, more general than groups, the number of semigroups and monoids have been counted up to order 9 and 10, respectively, using a combination of constraint satisfaction techniques implemented in the Minion constraint solver with bespoke symmetry breaking provided by the computer algebra system GAP [4,5,6].

In this paper we now consider the algebraic structures of finite Abel Grassmann Groupoids (AG-groupoids for short). AG-groupoids were first introduced by Naseeruddin and Kazim in 1972 [12] and have applications for example in the physics theory of flocks. They are generally considered midway between a groupoid and a commutative semigroup, that is, every commutative semigroup is an AG-groupoid but not vice versa. Thus AG-groupoids can also be non-associative, however they do not necessarily have the Latin square property. As a consequence neither of the enumeration techniques developed for quasigroups and loops or for semigroups and monoids can be employed directly. Our approach is based on the constraint solving technique developed for the enumeration of monoids and semigroups presented in [6]. However, since the original work explicitly exploited the associativity property for symmetry breaking we now present its novel adaptation to deal with our domain. Furthermore, we go beyond simple enumeration of the structures by the constraint solver and obtain a further division of the domain into interesting subclasses of AG-groupoids using the computer algebra system GAP. We have currently obtained enumeration results for AG-groupoids up to size 6 together with enumeration of some of the relevant subclasses. In addition, unlike in enumeration approaches using combinatorial techniques or algebraic counting, our enumeration also produces all multiplication tables for the structures found. These can be used both for further, more specialised, classification as well as be included into a library for GAP system in the future.

As no enumeration of AG-groupoids had been attempted before we present novel mathematical results, which are important as they give a first indication on the domain size of AG-groupoids as well as on the growth rate for the classes for increasing size of structures. This information can potentially be exploited when developing applications involving AG-groupoids. Our results also help to chart further the landscape of algebraic structures more general than groups.

The paper is organised as follows: in the next section we give an introduction to the mathematical theory of AG-groupoids. We then present an overview of the constraint solving techniques that we have used to enumerate AG-groupoids in Sec. 3, with a particular emphasis on their adaptations to the new domain and how symmetries are broken using the computer algebra system GAP. We then discuss the classification results in Sec. 4 before concluding in Sec. 5.

2 Background

We first give a brief introduction into the mathematical theory of AG-groupoids and especially define some of the properties we are interested in, for their classification.

AG-groupoids were introduced by Naseeruddin and Kazim in 1972 [12] and were originally called *left almost semigroup* (LA-semigroup). They have also been studied under the names *right modular groupoid* [2] and *left invertive groupoid* [10], before Stevanovic and Protic called the structure Abel-Grassman groupoids (or AG-groupoids for short) [34], which is the primary name under which they are known today. AG-groupoids generalise the concept of commutative semigroups and have an important application within the theory of flocks [25]. In addition to applications, a variety of properties have been studied for AG-groupoids and related structures. Mushtaq and Yusuf studied some basic properties of this structure [23] and introduced the notion of locally associative AG-groupoids [22]. Stevanovic and Protic introduced the notion of AG-bands and AG-3-bands in [34]. In [18] Mushtaq lifted the concepts of zeroids and idempoids from semigroups to AG-groupoids and some more weak associative laws were added in [19]. In more recent work the cancellativity of AG-groupoids was studied [31] and direct products of AG-groupoids have been introduced together with the notion of ideals and M-systems on AG-groupoids [21]. Furthermore, the structure of AG-groupoids has recently been fuzzified [14,13].

As an important subclass of AG-groupoids so called AG-groups where identified [20], which generalise the concept of Abelian groups. Some of their basic properties have subsequently been studied in [27] and their relationship with the algebraic structures of quasigroups and multiplication groups has been investigated in [30] and [29], respectively. AG-groups also have a geometrical interpretation that gives rise to their application in the context of parallelogram spaces [28].

However, in spite of investigations of AG-groupoids and their subclasses for nearly four decades no progress had been made in obtaining enumeration results. In fact, not even the exact number of non-associative AG-groupoids for the order 3, the smallest possible order, was known up to now.

We first recall that a groupoid is defined as a non-empty set S together with a binary operation $\circ : S \times S \to S$. In the remainder we will generally elide the binary operation. We now define an AG-groupoid as follows:

Definition 1. *Let S be a groupoid. S is called an AG-groupoid if for all $a, b, c \in S$ the following identity holds:*

$$(ab)c = (cb)a \tag{1}$$

Note that identity (1) is a reverse form of standard associativity.

Definition 2. *A groupoid S is called medial if S satisfies the medial law, that for all $a, b, c, d \in S$ we have*

$$(ab)(cd) = (ac)(bd). \tag{2}$$

It has been shown in [2], that every AG-groupoid is medial. Observe that the medial law is a property closely related to commutativity. Consequently, AG-groupoids can also be viewed as a generalisation of commutative semigroups and indeed one of our main classification of AG-groupoids will be to separate into associative and non-associative as well as commutative and non-commutative ones.

We now define a number of properties that give rise to interesting subclasses of AG-groupoids, which we identify in our classification.

Definition 3. *An AG-groupoid S is called weak associative if it satisfies the identity $(ab)c = b(ac)$ for all $a, b, c \in S$. We call S an AG*-groupoid.*

Definition 4. *An AG-groupoid S is called an AG-monoid if S has a left identity, i.e., there exists an element $e \in S$ such that for all elements $a \in S$ we have $ea = a$.*

Observe that since AG-groupoids are not necessarily associative, the existence of a left identity does not imply the existence of a general identity.

Definition 5. *An AG-groupoid S that satisfies the identity $a(bc) = b(ac)$ for any $a, b, c \in S$ is called an AG**-groupoid.*

Definition 6. *A groupoid S is called paramedial if S satisfies the paramedial law, i.e, for all $a, b, c, d \in S$ we have $(ab)(cd) = (db)(ca)$.*

One can easily verify the following facts that (i) every AG-monoid is an AG**-groupoid, and that (ii) every AG**-groupoid is paramedial. However, the converse is generally not true.

Example 1. As example structures we consider the four AG-groupoids of order 5 below, where (i) is an AG-monoid, (ii) is an AG*-groupoid (iii) is an AG**-groupoid and therefore also paramedial, while (iv) is a paramedial groupoid which is not an AG**-groupoid.

(i)

∘	0	1	2	3	4
0	0	0	0	0	0
1	0	0	3	0	1
2	0	1	2	3	4
3	0	0	1	0	3
4	0	3	4	1	2

(ii)

∘	0	1	2	3	4
0	4	1	1	2	4
1	1	1	1	1	1
2	1	1	1	1	1
3	1	1	1	2	1
4	4	1	1	1	4

(iii)

∘	0	1	2	3	4
0	1	0	4	4	3
1	4	3	1	1	0
2	0	1	2	3	4
3	0	1	3	3	4
4	3	4	0	0	1

(iv)

∘	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	1	0	4	4	1
3	0	1	4	4	0
4	0	0	0	0	0

Definition 7. *Let S be an AG-groupoid. We call S locally associative if for all $a \in S$ $a(aa) = (aa)a$ holds.*

Definition 8. *An AG-groupoid S is called an AG-2-band or simply AG-band if for all $a \in S$ the identity $aa = a$ is satisfied. In other words in an AG-band S every element is idempotent.*

Definition 9. *An AG-groupoid S is called an AG-3-band if for every $a \in S$ we have $a(aa) = (aa)a = a$.*

Definition 10. *An element a of an AG-groupoid S is called left cancellative if $ax = ay \Rightarrow x = y$ for all $x, y \in S$. Similarly an element a of an AG-groupoid S is called right cancellative if $xa = ya \Rightarrow x = y$ for all $x, y \in S$. An element a of an AG-groupoid S is called cancellative if it is both left and right cancellative. S is called (left,right) cancellative if all of its elements are (left,right) cancellative and is called cancellative if it is both left and right cancellative.*

This definition gives rise to the following theorem, which we state without proof:

Theorem 1. *Let S be an AG-groupoid, the following is equivalent:*

(i) S is left cancellative.
(ii) S is right cancellative.
(iii) S is cancellative.

Finally we define AG-groups, an important subclass of AG-groupoids, which generalises the concept of inverses.

Definition 11. *An AG-groupoid S is called an AG-group if S has a left identity $e \in S$ and inverses with respect to this identity, i.e., for all elements $a \in S$ there exists an element $b \in S$ such that $ab = ba = e$.*

It has been shown in [27], that every AG-group G with local associativity is an Abelian group.

Example 2. As further example structures we give (i) an AG-band, (ii) an AG-3-band, and (iii) an AG-group below. For the latter we observe that the element 3 is the left identity element and that the structure is indeed not a group as there is no corresponding right identity.

(i)

∘	0	1	2	3	4
0	0	3	1	4	1
1	4	1	3	0	3
2	3	0	2	1	4
3	1	4	0	3	0
4	3	0	4	1	4

(ii)

∘	0	1	2	3	4
0	4	3	0	2	1
1	2	4	3	1	0
2	1	2	4	0	3
3	3	0	1	4	2
4	0	1	2	3	4

(iii)

∘	0	1	2	3	4
0	3	0	1	4	2
1	4	3	0	2	1
2	2	4	3	1	0
3	0	1	2	3	4
4	1	2	4	0	3

3 Constructing AG-Groupoids

To obtain all AG-groupoids up to isomorphism we adapt a method which was introduced in [5] and [6] in the search for monoids. The idea is to combine the advantages of a constraint solver for a fast search with that of a computer algebra system to efficiently rule out isomorphic copies. We will give a brief overview of the used technique. More detailed explanations and applications for various subclasses of semigroups can also be found in [4, Chapters 4, 5].

There are a number of important differences in the approach presented here to the one developed for enumerating semigroups and monoids. Instead of one search for each order, several independent searches were performed in [5,4,6], and for many of them it became far easier to avoid isomorphic solutions. These case splits were often based on structural knowledge about monoids and semigroups which also lead to a more efficient search in the remaining difficult, but more specific, cases. The enumeration of monoids and semigroups also benefit hugely from the fact that not all such objects needed to be counted. For semigroups there exists a formula for the majority of such objects [4, Section 2.3], while most monoids were constructed using semigroups and groups of lower order. For AG-groupoids we performed only one search for each order, as attempts to accelerate the search using a case split were not successful. This might change in the future, when the mathematical understanding of AG-groupoids has deepened further.

3.1 CSP and Minion

Constraint Programming is a powerful technique for solving large-scale combinatorial problems. To get an overview of this area the reader might want to start with [26]. Here we provide just basic definitions meeting our needs.

Definition 12. *A* constraint satisfaction problem (CSP) *is a triple* (V, D, C)*, consisting of a finite set* V *of* variables*, a finite set* D*, called the* domain*, of* values*, and a set* C *containing subsets of* D^V *(that is, all functions from* V *to* D*) called* constraints*.*

In practice, constraints, instead of being subsets of D^V, are usually formulated as conditions uniquely defining such subsets. Intuitively it is clear that one is looking for assignments of values in the domain of a CSP to all variables such that no constraint is violated. This is formalised in the next definition.

Definition 13. *Let $L = (V, D, C)$ be a CSP. A partial function $f : V \to D$ is an* instantiation. *An instantiation f* satisfies *a constraint if there exists a function F in the constraint, such that $F(x) = f(x)$ for all $x \in V$ on which f is defined. An instantiation is* valid *if it satisfies all the constraints in C. An instantiation defined on all variables is a* total *instantiation. A valid, total instantiation is a* solution *to L.*

The class of CSPs is a generalisation of propositional satisfiability (SAT), and is therefore NP-complete. Solving problems using CSPs proceeds in two steps: modelling and solving. Solvers typically proceed by building a search tree, in which the nodes are assignments of values to variables and the edges lead to assignment choices for the next variable. If at any node a constraint is violated, then search backtracks. If a leaf is reached, then no constraints are violated, and the assignments provide a solution.

For our purposes we rely on Minion [9] as solver which offers fast, scalable constraint solving. A major feature of modern SAT solvers is their optimised use of modern computer architecture. Using this approach, Minion has been designed to minimise memory usage.

3.2 Symmetry Breaking and GAP

CSPs are often highly symmetric. Given any solution, there can be others which are equivalent in terms of the underlying problem. Symmetries may be inherent in the problem, or be created in the process of representing the problem as a CSP. Without symmetry breaking (henceforth SB), many symmetrically equivalent solutions may be found and, often more importantly, many symmetrically equivalent parts of the search tree will be explored by the solver. An SB method aims to avoid both of these problems.

Definition 14. *Let $L = (V, D, C)$ be a CSP.*

1. *Elements in the set $V \times D$ are called* literals. *Literals are denoted in the form $(x = k)$ with $x \in V$ and $k \in D$.*
2. *Let χ denote the set of all literals of L. A permutation $\pi \in S_\chi$ is a* symmetry *of L if, under the induced action on subsets of χ, instantiations are mapped to instantiations and solutions to solutions.*
3. *A* variable-value symmetry *is a symmetry $\pi \in S_\chi$ such that there exists an element (τ, δ) in $S_V \times S_D$ with $(x = k)^\pi = (x^\tau = k^\delta)$ for all $(x = k) \in \chi$.*

The given definition of symmetry of a CSP is relatively strong. On the other hand all symmetries are variable-value symmetries in our case and as such will always send instantiations to instantiations. For more information on symmetries in CSPs, including different definitions see [26, Chapter 10].

There is a general technique, called lex-leader, for generating constraints that break symmetries [3]. The idea of lex-leader is to order solutions by defining an order on the literals of the CSP. This allows one to define the canonical representative in each set of symmetric solutions to be the solution which is

smallest (or largest) with respect to the order. To define an order on solutions of a CSP L, first fix an ordering $(\chi_1, \chi_2, \ldots, \chi_{|V||D|})$ of the literals $\chi = V \times D$. Given the fixed ordering of the literals, an instantiation can be represented as a bit vector of length $|V||D|$. The bit in the i-th position is 1 if χ_i is contained in the instantiation and otherwise the bit is 0. The bit vector for the instantiation $I \subseteq \chi$ corresponding to the ordering of the literals $(\chi_1, \chi_2, \ldots, \chi_{|V||D|})$ will be denoted by $(\chi_1, \chi_2, \ldots, \chi_{|V||D|})_I$. Of all bit vectors corresponding to a set of symmetric solutions of L, one is the lexicographic maximal, which shall be the property identifying the canonical solution. If $\geq_{\text{lex}}$ denotes the standard lexicographic order on vectors, extend L by adding for all symmetries π the constraint

$$(\chi_1, \chi_2, \ldots, \chi_{|V||D|})_I \geq_{\text{lex}} (\chi_1^\pi, \chi_2^\pi, \ldots, \chi_{|V||D|}^\pi)_I. \tag{3}$$

Then, from each set of symmetric solutions in L, exactly those with lexicographic greatest bit vector are solutions of the extended CSP.

To generate the constraints for symmetry breaking we use specialist software that provides robust, efficient and extensive implementations of algorithms in abstract algebra. GAP [8] (Groups, Algorithms and Programming) is a system for computational discrete algebra with particular emphasis on, but not restricted to, computational group theory. GAP provides a large library of functions that implement algebraic algorithms.

3.3 Search

We first formulate a CSP to search for all different AG-groupoids on the set $\{1, 2, \ldots, n\}$ for a positive integer n, and will subsequently add symmetry breaking to it.

CSP 1. *For a positive integer n define a CSP $L_n = (V_n, D_n, C_n)$. The set V_n consists of n^2 variables $\{A_{i,j} \mid 1 \leq i, j \leq n\}$, one for each position in an $(n \times n)$-multiplication table, having domain $D_n = \{1, 2, \ldots, n\}$. The constraints in C_n are*

$$A_{A_{i,j},k} = A_{A_{k,j},i} \text{ for all } i, j, k \in \{1, 2, \ldots, n\}, \tag{4}$$

reflecting the left-invertive law.

In Minion the constraint (4) is enforced using element constraints. The constraint $element(vector, i, val)$ specifies that, in any solution, $vector[i] = val$. We add a new variable $T_{a,b,c}$ for each triple (a, b, c). The pair of constraints

$$element(column(c), A_{a,b}, T_{a,b,c}) \text{ and } element(column(a), A_{c,b}, T_{a,b,c})$$

then enforces (4) for the triple.

As mentioned in Section 3.2, modelling a problem often introduces symmetries. In our case this happens by introducing identifiers, 1 up to n, for the n elements, even though we want them to be initially indistinguishable. The symmetries are the isomorphism between AG-groupoids, hence elements in S_n. To find a single representative from every equivalence class we have to break

Table 1. Solutions and timings for L_n and $\overline{L}_n$

Order n	1	2	3	4	5	6
L_n, solutions	1	6	105	7 336	3 756 645	28 812 382 776
–, solve time	ϵ	ϵ	ϵ	ϵ	25 s	104 245 s
$\overline{L}_n$, solutions	1	3	20	331	31 913	40 104 513
–, solve time	ϵ	ϵ	ϵ	ϵ	ϵ	121 s

The times are rounded to seconds. They were obtained using version 0.11 of Minion on a machine with 2.80 GHz Intel X-5560 processor. The symbol ϵ stands for a time less than 0.5 s.

these introduced symmetries. We want to use the lex-leader method described in Section 3.2 and therefore define an ordering of the literals. We use

$$(A_{1,1}, A_{1,2}, \ldots, A_{1,n}, A_{2,1}, \ldots, A_{2,n}, \ldots, A_{n,1}, \ldots, A_{n,n}) \tag{5}$$

as variable order and define for the literals $(A_{i,j} = k) \leq_{\text{lex}} (A_{r,s} = t)$ if either $A_{i,j}$ comes earlier than $A_{r,s}$ in (5) or $k \leq t$. The canonical table in every isomorphism class is then defined by having the lex-largest bit vector with respect to this ordering of literals, which corresponds to the lex-smallest table with respect to standard row-by-row ordering. By adding for each $\pi \in S_n$ constraint (3) to L_n we obtain a CSP $\overline{L}_n$ which has AG-groupoids of order n up to isomorphism as solutions.

The data from running the instances L_n and $\overline{L}_n$ for $1 \leq n \leq 6$ is summarised in Table 1. Further classification for non-isomorphic AG-groupoids is presented in the following section. A computation solving $\overline{L}_7$ to enumerate AG-groupoids of order 7 up to isomorphism is currently running and counted already more than $3 \cdot 10^{11}$ solutions in two weeks.

4 Classification of AG-Groupoids

To obtain our classification results we have used Minion as discussed in the previous section to enumerate the entire space of non-isomorphic AG-groupoids as well as to produce the multiplication tables for all structures. This data is then further exploited to perform more fine-grained division into subclasses of AG-groupoids with respect to the properties presented in Sec. 2. This classification task is performed in GAP using functionality built on top of GAP's Loops library [24]. Although for large data sets (e.g, in the case of the classification of AG-groupoids of order 6) we use some parallelisation, this is fairly trivial and we will not present details here.

Table 2 presents our main result, the enumeration of the total number of AG-groupoids of orders 3 to 6, which have not been known to date. These numbers are further broken down with respect to associativity and commutativity properties of the AG-groupoids. Observe that we only consider three classes here as it

Table 2. General classification result for AG-groupoids of orders 3–6

Order	3	4	5	6
Total	20	331	31913	40104513
Associative & commutative	12	58	325	2143
Associative & non-commutative	0	4	121	5367
Non-associative & non-commutative	8	269	31467	40097003

can be easily shown that every AG-groupoid that is non-associative is also non-commutative.

Tables 3, 4 and 5 then present the results of our further classification into subclasses. Observe that the structures in Table 5 are all commutative semigroups, which is the type of structure generalised by the notion of AG-groupoid. As a consequence some of the classification results are expected to be trivial as it is know that some of the properties are exhibited by all structures in that class. For example, every associative structure is also locally associative, and every semigroup is paramedial.

Similarly, in Table 4 it was to be expected that some of the rows would be empty or contain all structures in the class. However, it should be pointed out that it was assumed that all semigroups would also be AG*- and AG**-groupoids. Yet the result of our classification clearly shows that for order 6 there has to exist a further class of non-commutative semigroups that are neither AG*- nor AG**-groupoids. This is a novel, non-trivial result from our classification which will lead to a new class of structures to be investigated in the future.

A further minimal example, which was not known before, appears in Table 3. We see that there are two non-associative AG-3-bands of order 6 which are not AG-bands. This is of particular interest since it concerns AG-groupoids which are not semigroups.

Regarding the correctness of our results we are very confident that there are no errors. We only utilised established and well-tested functionality in GAP and

Table 3. Classification of non-associative AG-groupoids

Order	3	4	5	6
Total	8	269	31467	40097003
AG-monoids	1	6	29	188
AG*-groupoid	0	0	0	9
AG**-groupoid	4	39	526	13497
Locally Associative	3	78	4482	1818828
AG-band	0	1	3	8
AG-3-band	0	1	3	10
Paramedial	8	264	31006	39963244
Cancellative	1	4	4	1
AG-groups	1	2	1	1

Table 4. Classification of associative, non-commutative AG-groupoids

Order	3	4	5	6
Total	0	4	121	5367
AG-monoids	0	0	0	0
AG*-groupoid	0	4	121	5360
AG**-groupoid	0	4	121	5360
Locally Associative	0	4	121	5367
AG-band	0	0	0	0
AG-3-band	0	0	0	0
Paramedial	0	4	121	5367
Cancellative	0	0	0	0
AG-groups	0	0	0	0

Table 5. Classification of associative, commutative AG-groupoids

Order	3	4	5	6
Total	12	58	325	2143
AG-monoids	5	19	78	421
AG*-groupoid	12	58	325	2143
AG**-groupoid	12	58	325	2143
Locally Associative	12	58	325	2143
AG-band	2	5	15	53
AG-3-band	4	13	41	162
Paramedial	12	58	325	2143
Cancellative	1	2	1	1
AG-groups	1	2	1	1

Minion, and a more involved version of our approach has successfully been used before [4,5,6]. Moreover, most of the results were verified using other means. For AG-monoids we compared the numbers with a purely algebraic way of counting developed by Sergey Shpectorov and the second author. (A publication containing a description of the counting method is in preparation.) In the case of associative AG-groupoids, that is for all results presented in Tables 4 and 5, we obtained identical numbers from experiments with the GAP package Smallsemi [7], which contains a database of all semigroups up to order 8. Finally, all numbers less than one million in Tables 2 to 5 were verified by one of the reviewers using Mace4 and Isofilter (parts of the Prover9/LADR package) [15].

5 Conclusions

We have presented novel classification results for the algebraic domain of AG-groupoids. We have produced both enumeration results for orders up to 6 and a partial classification of the domain using additional algebraic properties. To obtain these results we have employed a combination of the constraint solver

Minion and the computer algebra system GAP. Thereby GAP is used on the one hand to perform symmetry breaking during the constraint solving process and on the other hand to perform the subsequent subclassification. While Minion has been previously applied in the classification of Semigroups and Monoids, the adaptation to our new application domain as discussed in Sec. 3 are both novel and non-trivial.

One of the advantages of our classification approach over techniques that use combinatorial or algebraic considerations for enumeration, is that it allows us to produce the multiplication tables of the structures under consideration. These can be further used to produce more fine-grained subclassifications as we have done in the case of AG-groupoids via a two step approach: firstly separating with respect to associativity and commutativity properties followed by a second refinement step using more specialised properties. While these were primarily motivated by mathematical considerations, the obtained results have already stimulated further investigations into other properties that could help to further subdivide the domain and lead to interesting, distinct classes of algebraic structures. For example our classification results of order 6 AG-groupoids have already yielded a heretofore unknown subclass of non-commutative semigroups that are neither AG*- nor AG**-groupoids as well as a new subclass of non-associative AG-groupoids that are AG-3-bands but not AG-bands. We hope to obtain more evidence on these new classes once our classification of order 7 has concluded and will then start its theoretical investigation.

In general, we see the work presented in this paper as an important stepping stone for a further more fine-grained charting for algebraic structures that are more general than groups. Having the computational means to obtain reliable, non-trivial classification results will play a crucial role in this endeavour. As a consequence we intend to collate both our data and the functionality we have implemented in the process of our investigations into a GAP package to be published shortly.

Acknowledgements

We like to thank the anonymous reviewers and are especially thankful to the reviewer who verified most of our results using Mace4 and Isofilter and pointed out two mistakes in the numbers of non-associative AG-3-bands in an earlier draft of this paper.

References

1. Ali, A., Slaney, J.: Counting loops with the inverse property. Quasigroups and Related Systems 16(1), 13–16 (2008)
2. Cho, J.R., Pusan, Jezek, J., Kepka, T.: Paramedial groupoids. Czechoslovak Mathematical Journal 49(124) (1996)
3. Crawford, J.M., Ginsberg, M.L., Luks, E.M., Roy, A.: Symmetry-breaking predicates for search problems. In: Proc. of KR 1996, pp. 148–159. Morgan Kaufmann, San Francisco (1996)

4. Distler, A.: Classification and Enumeration of Finite Semigroups. PhD thesis, University of St Andrews (2010)
5. Distler, A., Kelsey, T.: The monoids of order eight and nine. In: Autexier, S., Campbell, J., Rubio, J., Sorge, V., Suzuki, M., Wiedijk, F. (eds.) AISC 2008, Calculemus 2008, and MKM 2008. LNCS (LNAI), vol. 5144, pp. 61–76. Springer, Heidelberg (2008)
6. Distler, A., Kelsey, T.: The monoids of orders eight, nine & ten. Ann. Math. Artif. Intell. 56(1), 3–21 (2009)
7. Distler, A., Mitchell, J.D.: Smallsemi - A library of small semigroups. A GAP 4 package [8], Version 0.6.2 (2010), http://tinyurl.com/jdmitchell/smallsemi
8. The GAP Group, GAP – Groups, Algorithms, and Programming, Version 4.4.12 (2008), http://www.gap-system.org
9. Gent, I.P., Jefferson, C., Miguel, I.: Minion: A fast scalable constraint solver. In: Proc. of ECAI 2006, pp. 98–102. IOS Press, Amsterdam (2006)
10. Holgate, P.: Groupoids satisfying a simple invertive law. Math. Student 61, 101–106 (1992)
11. Humphreys, J.: A Course in Group Theory. Oxford University Press, Oxford (1996)
12. Kazim, M.A., Naseerudin, M.: On almost semigroups. Alig. Bull. Math. 2, 1–7 (1972)
13. Khan, A., Jun, Y.B., Mahmood, T.: Generalized fuzzy interior ideals in Abel Grassmann's groupoids. Mathematics and Mathematical Sciences (2010)
14. Khan, M., Nouman, M., Khan, A.: On fuzzy Abel Grassmann's groupoids. Advances in Fuzzy Mathematics 5(3), 349–360 (2010)
15. McCune, W.: Mace4 Reference Manual and Guide. Mathematics and Computer Science Division, Argonne National Laboratory, ANL/MCS-TM-264 (August 2003)
16. McKay, B.D., Meynert, A., Myrvold, W.: Small Latin squares, quasigroups and loops. J. of Combinatorial Designs 15, 98–119 (2007)
17. McKay, B.D., Wanless, I.M.: On the number of Latin squares. Ann. Combin. 9, 335–344 (2005)
18. Mushtaq, Q.: Zeroids and idempoids in AG-groupoids. Quasigroups and Related Systems 11, 79–84 (2004)
19. Mushtaq, Q., Kamran, M.S.: On LA-semigroups with weak associative law. Scientific Khyber 1, 69–71 (1989)
20. Mushtaq, Q., Kamran, M.S.: Left almost groups. Proc. Pak. Acad. of Sciences 33, 1–2 (1996)
21. Mushtaq, Q., Khan, M.: Direct product of Abel Grassmann's groupoids. Journal of Interdisciplinary Mathematics 11(4), 461–467 (2008)
22. Mushtaq, Q., Yusuf, S.M.: On locally associative LA-semigroups. J. Nat. Sci. Math XIX(1), 57–62 (1979)
23. Mushtaq, Q., Yusuf, S.M.: On LA-semigroups. Alig. Bull. Math. 8, 65–70 (1978)
24. Nagy, G.P., Vojtechovsky, P.: LOOPS: Computing with quasigroups and loops in GAP v1.0, computational package for GAP, http://www.math.du.edu/loop
25. Naseeruddin, M.: Some studies on almost semigroups and flocks. PhD thesis, The Aligarg Muslim University, India (1970)
26. Rossi, F., van Beek, P., Walsh, T.: Handbook of Constraint Programming (Foundations of Artificial Intelligence). Elsevier Science Inc., Amsterdam (2006)
27. Shah, M., Ali, A.: Some structural properties of AG-groups. Int. Mathematical Forum 6(34), 1617–1661 (2011)
28. Shah, M., Sorge, V., Ali, A.: A study of AG-groups as parallelogram spaces (submitted)

29. Shah, M., Ali, A., Sorge, V.: Multiplication group of an AG-group (submitted)
30. Shah, M., Ali, A., Sorge, V.: A study of AG-groups as quasigroups (submitted)
31. Shah, M., Shah, T., Ali, A.: On the cancellativity of AG-groupoids. International Journal Of Algebra (to appear, 2011)
32. Slaney, J.: FINDER, Notes and Guide. Center for Information Science Research, Australian National University (1995)
33. Sorge, V., Meier, A., McCasland, R., Colton, S.: Classification results in quasigroup and loop theory via a combination of automated reasoning tools. Comm. Univ. Math. Carolinae 49(2-3), 319–340 (2008)
34. Stevanovic, N., Protic, P.V.: Composition of Abel-Grassmann's 3-bands. Novi Sad J. Math. 34(2), 175–182 (2004)

Retargeting OpenAxiom to Poly/ML: Towards an Integrated Proof Assistants and Computer Algebra System Framework

Gabriel Dos Reis[1], David Matthews[2], and Yue Li[1]

[1] Texas A&M University,
College Station, USA
[2] Prolingua Ltd,
Edinburgh, Scotland

Abstract. This paper presents an ongoing effort to integrate the AXIOM family of computer algebra systems with Poly/ML-based proof assistants in the same framework. A long-term goal is to make a large set of efficient implementations of algebraic algorithms available to popular proof assistants, and also to bring the power of mechanized formal verification to a family of strongly typed computer algebra systems at a modest cost. Our approach is based on retargeting the code generator of the OpenAxiom compiler to the Poly/ML abstract machine.

Keywords: OpenAxiom, Poly/ML-based proof assistants, runtime systems.

1 Introduction

Computer algebra systems seek efficient implementations of algebraic algorithms. As it happens, they occasionally "cut corners", making assumptions that are not always obvious from official documentations. Proof assistants, on the other hand, focus on mechanized proofs; they are uncompromising on formal correctness at the expense of efficiency. In this paper, we report on an ongoing effort to integrate OpenAxiom [19], a member of AXIOM family of strongly-typed computer algebra systems, with proof assistants (notably Isabelle/HOL [12]) based on the Poly/ML [20] programming system.

A large body of work [3,5,4,11,2,6,10] exists on interfacing computer algebra systems with logical frameworks. Most establish external communication protocols, links, and external exchange formats between proof assistants and computer algebra systems. Many depict those communications in the model of master-slave, where the master is the proof assistant and the slave is the computer algebra system. The work reported in this paper is novel in several aspects and defies assumptions from previous work.

First, we are considering a *strongly typed* computer algebra system. For concreteness, our work is done with OpenAxiom, a member of the AXIOM family. A distinctive feature of AXIOM is that every single computation is performed in a particular environment called *domain of computation*. A domain provides representations for values and operations for manipulating them. In AXIOM, and unlike in most popular computer algebra systems, a value is essentially useless and meaningless without knowledge of the intended domain — i.e. how to interpret that value. Domains, in turn, satisfy various specifications defined by *categories*. For obvious reasons, categories are organized into

J.H. Davenport et al. (Eds.): Calculemus/MKM 2011, LNAI 6824, pp. 15–29, 2011.

hierarchies, mirroring development of mathematical concepts and knowledge. In practice, domains and categories are huge data structures, usually with non-trivial cyclical dependencies. For example, the domain `Integer` of integer values satisfy the category `Ring` that specifies a ring structure over the common addition and multiplication operators. On the other hand, the specification of `Ring` involves the domain `Integer` — if anything, to specify the meaning of the expression $n \cdot x$ which denotes the addition of an arbitrary a ring element x to itself n times, and also the natural injection of integers into any ring structure. Furthermore, a simple computation such as `factor(x^2-2*x+1)` has the OpenAxiom system load no less than 19 domains or packages, not counting the hundreds of pre-loaded domains. Clearly, under these circumstances, devising a communication protocol for externalizing enough information to faithfully reconstruct the result of a computation, or check the validity of computational steps can be very inefficient in practice. Day-to-day experiences with dumping OpenAxiom domain and category databases or debug information suggest negative impacts on other support services such as garbage collection.

Second, this project — unlike most existing work — envisions a symbiotic coexistence of OpenAxiom and a Poly/ML-based proof assistant. That is, we envision a tight integration of OpenAxiom and, say, Isabelle/HOL where both systems run in the same address space with Isabelle/HOL calling on OpenAxiom for algebraic computation and OpenAxiom calling on Isabelle/HOL for validating computational steps or performing logical deduction. Achieving this tight integrating requires a formal account of both OpenAxiom program representation (Sec. 3) and the Poly/ML runtime system (Sec. 4), and also a formal correspondence between both (Sec. 5.) The challenge is compounded by the fact that neither the Spad programming language nor the Poly/ML abstract machine have a formal specification. A key contribution of this paper is a step toward formal specification of the Poly/ML abstract machine language and translation algorithms. We anticipate that these formal accounts will provide foundation for further scrutiny and other research projects such as an end-to-end verification of OpenAxiom libraries, or a complete mechanical verification of the Poly/ML system — the runtime system underlying many popular logical framework implementations.

Third, this project is also an excellent opportunity to clarify the semantics of the AXIOM family system. Because of historical implementation artifacts, it is sometimes mistakenly believed that AXIOM is a Lisp system. While it is true that AXIOM systems currently use Lisp systems as base runtime systems, they actually do have type-erasure semantics in the style of ML [7] — except for runtime queries of category satisfaction, but the semantics of those queries are essentially intensional in nature and are compilable with a type-erasure semantics. This project of retargeting OpenAxiom to Poly/ML runtime system clarifies that subtle but crucial aspect of AXIOM systems and offers a validation of that theory.

Finally, this work enables cross-cutting technology reuse. For example, retargeting OpenAxiom to Poly/ML makes available (at virtually no cost) the recent concurrency work [17] done in Poly/ML that has been so beneficial to the Isabelle framework to take advantage of multi-core machines. Furthermore, retargeting the Poly/ML platform opens the possibility of concurrent validation of OpenAxiom computations by several Poly/ML-based proof assistants.

2 The OpenAxiom and Poly/ML Systems

2.1 OpenAxiom

OpenAxiom [19] is an evolution of the AXIOM [13] computer algebra system. It is equipped with a strongly-typed programming language (named Spad) for writing large scale libraries, and a scripting language for interactive uses and for programming in the small. The Spad programming language features a two-level type system to support data abstraction and generic structures and generic algorithms.

Domain of computation. A central tenet of AXIOM philosophy is that computations occur in a given domain of discourse or *domain of computation*. For instances, the objects $(X^2 - 1)/(X - 1)$ and $X + 1$ are equal in $\mathbf{Q}(X)$, but not as functions from $\mathbf{R}$ to $\mathbf{R}$; see the excellent analysis of this intricate problem by Davenport [9] for further details.

Categories as specifications. Since several domains of computations may implement the same specification — *e.g.* both `Integer` and `String` implement equality comparison — the Spad programming language provides abstraction tools to write specifications as first class objects: *categories*. Here is a specification for domains of computations that implement equality comparison:

```
BasicType: Category == Type with
    =: (%,%) -> Boolean
    ~=: (%,%) -> Boolean
```

Categories can extend other categories, or may be parameterized, or can be composed out of previously defined categories. For example, a semi-group is an extension of `BasicType` with the requirement of an associative operator named `*`:

```
SemiGroup: Category == BasicType with
   *: (%,%) -> %
```

and a left-linear set over a semi-group `S` is a set that is stable by "multiplication" or dilation by values in `S`:

```
LeftLinearSet(S: SemiGroup): Category == Basictype with
    *: (S,%) -> %
```

We similarly define the notion of right-linear. A linear set over a semi-group `S` is a set that is both left-linear and right-linear over `S`:

```
LinearSet(S: SemiGroup): Category ==
   Join(LeftLinearSet S, RightLinearSet S)
```

More complex algebraic structures are specified using similar composition and extension techniques.

2.2 Poly/ML

Poly/ML is a system originally written by the second author while he was in the Computer Laboratory at Cambridge University. Poly/ML was initially developed as an experimental language, Poly, similar to ML but with a different type system. Among

the first users was Larry Paulson who used it to develop the Isabelle theorem prover. More recently David Matthews has continued to develop Poly/ML. The Standard Basis Library has been implemented and the compiler converted to the 1997 Definition of Standard ML (Revised). Poly/ML is available for the most popular architectures and operating systems. There are native code versions for the i386 (32 and 64 bit), Power PC and Sparc architectures. There is a byte-code interpreted version which can be used on unsupported architectures.

3 Spad Program Representation

The Spad programming language is strongly typed. Yet, it allows for runtime instantiation of domains and categories. Consequently, categories and domains are both compile-time and runtime objects. From now on, we will discuss only the representation of category objects. Domains and packages are similarly represented, with some variations to attend to data specific to domains.

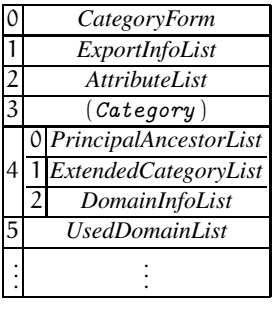


Fig. 1. Layout of category objects

Category object layout. A category object is represented [8] as a large heterogeneous tuple as shown in Fig. 1. Its components have the following meaning:

- slot 0 holds the canonical category form of the expression whose evaluation produces the category object under consideration
- slot 1 holds a list of function signatures exported by the category
- slot 2 holds a list attributes and the condition under which they hold
- slot 3 always contain the form (*Category*). It serves as a runtime type checking tag
- slot 4 contains three parts:
  - a list of principal ancestor category forms
  - a list of directly extended category forms
  - a list of domain explicitly used in that category
- slot 5 holds the list of all domain forms mentioned in the exported signatures.
- each of the slots 6 (and onwards) holds either a runtime information about a specific exported signature, or a pointer to domain object or a category object.

Because several existing OpenAxiom libraries assume the above layout, translation from Spad to the Poly/ML must preserve its observable behavior.

3.1 OIL: OpenAxiom Intermediate Language

Traditionally, AXIOM compilers translate Spad programs to Lisp, then compile the generated Lisp code. The OpenAxiom compiler has been modified to generate an intermediate representation that is independent of Lisp, to enable re-targeting to several backends. The intermediate language, called OIL, is a lambda calculus with constants and shown in Fig. 2.

Module	m	$::= \overrightarrow{d}$
Definition	d	$::= (\texttt{def}\ x\ e)$
expression	e,p	$::= v \mid l \mid [\overrightarrow{e}] \mid d \mid q \mid (f\ \overrightarrow{e}) \mid (\texttt{when}\ (\overrightarrow{w})) \mid (\texttt{bind}\ (\overrightarrow{(x\ e)})\ e)$ $\mid (\texttt{lambda}\ (\overrightarrow{x})\ e) \mid (\texttt{store}\ l\ e) \mid (\texttt{loop}\ (\overrightarrow{i})\ e) \mid (\texttt{seq}\ \overrightarrow{e})$
Location	l	$::= x \mid (\texttt{tref}\ x\ n)$
Branches	w	$::= (p\ e) \mid (\texttt{otherwise}\ e)$
Iterators	i	$::= (\texttt{step}\ x\ e\ e\ e) \mid (\texttt{while}\ p) \mid (\texttt{until}\ p) \mid (\texttt{suchthat}\ x\ e)$
Domain form	t	$::= x \mid (\texttt{D}\ \overrightarrow{t})$
Category form	c	$::= (\texttt{C}\ \overrightarrow{t}) \mid (\texttt{Join}\ \overrightarrow{c}) \mid (\texttt{mkCategory}\ k\ [\overrightarrow{[\sigma,q]}]\ [\overrightarrow{a}]\ [\overrightarrow{t}]\ \texttt{nil})$
Query	q	$::= b \mid (\texttt{hascat}\ t\ c) \mid (\texttt{hassig}\ t\ \sigma) \mid (\texttt{hasatt}\ t\ a)$ $\mid (\texttt{not}\ q) \mid (\texttt{and}\ q\ q) \mid (\texttt{or}\ q\ q)$
Constructor kind	k	$::= \texttt{category} \mid \texttt{domain} \mid \texttt{package}$
Signature	σ	$::= [x,t,\overrightarrow{t}]$
Builtin operator	o	$::= \texttt{eq} \mid \texttt{and} \mid \texttt{or} \mid \texttt{not} \mid \texttt{iadd} \mid \texttt{isub} \mid \ldots$
Function name	f	$::= o \mid \texttt{C} \mid \texttt{D} \mid x$
Literal values	v	$::= \texttt{nil} \mid b \mid n \mid s$
Boolean literal	b	$::= \texttt{false} \mid \texttt{true}$
Integer literal	n	
String literal	s	
Domain constructor	D	
Category constructor	C	
Attribute	a	
Identifier	x	

Fig. 2. OpenAxiom Intermediate Language

A module is a collection of top-level definitions. The body of a definition can be a lambda expression, a conditional expression, or a binding of local variables in an expression. We include some builtin operators (*e.g.*for addition on integers, category composition, etc.)

Example. Here is how the intermediate representation of the `BasicType` category from Sec. 2.1 looks:

```
(def BasicType;AL nil)

(def BasicType;
  (lambda ()
    (bind ((g (Join (Type)
```

```
                    (mkCategory domain
                        '(((= ((Boolean) $ $)) true)
                          ((~= ((Boolean) $ $)) true)
                          ((before? ((Boolean) $ $)) true))
                       nil '((Boolean)) nil))))
        (store (tref g 0) '(BasicType))
        g)))

(def BasicType
  (lambda ()
    (when ((not (eq BasicType;AL nil)) BasicType;AL)
          (otherwise (store BasicType;AL (BasicType;))))))
```

The compilation of a category typically produces tree top-level definitions: the definition of a cache holding instantiations of the category, the definition of a "worker" function that actually produces a category object for first-time instantiations, and a "wrapper" function around the worker function. The wrapper function is what the system invokes when a category is instantiated. In this example, *BasicType;AL* is the cache, *BasicType;* is the worker function, and *BasicType* is the wrapper function.

4 Poly/ML Codetrees

The front-end of the Poly/ML compiler performs syntax- and type-checking and produces an intermediate code in the form of a codetree. This untyped representation is machine-independent and, after optimization, is used to produce the machine-dependent code to execute. Our compilation process for Spad generates this code tree. The codetree is an ML data structure and does not have a canonical text representation. For the purposes of explanation the structure can be approximated by the grammar in Fig. 3.

Variables are given numerical names. For instance, a declaration $\mathrm{DECL}(\mathtt{k}, e)$ has a number $\mathtt{k}$ and a codetree term e. The codetree e denotes an expression to be bound to the variable $\mathtt{k}$. This can then be referenced within the rest of the containing block by means of a $\mathrm{LOCAL}(\mathtt{n}, \mathtt{k})$ codetree term. The number $\mathtt{k}$ corresponds to the identifier used in the declaration and $\mathtt{n}$ is the "nesting depth": zero if the reference is within the same function and non-zero if it is within an inner function. Function parameters are

Declaration	$\mathtt{d}$	$::= \mathrm{DECL}(\mathtt{k}, e)$
Code tree	e,$\mathtt{p}$	$::= \mathrm{NIL} \mid \mathtt{c} \mid \mathtt{d} \mid \mathrm{LIT}(\mathtt{n}) \mid \mathrm{ADDRESS}(\mathtt{a}) \mid \mathrm{BUILTIN}(\mathtt{r}) \mid \mathrm{EVAL}(e, [\overrightarrow{e}])$
		$\mid \mathrm{IF}(\mathtt{p}, e, e) \mid \mathrm{BLOCK}([\overrightarrow{e}]) \mid \mathrm{INDIRECT}(\mathtt{n}, e) \mid \mathrm{RECORD}([\overrightarrow{e}])$
		$\mid \mathrm{BEGINLOOP}([\overrightarrow{\mathtt{d}}], e) \mid \mathrm{LOOP}([\overrightarrow{e}]) \mid \mathrm{LAMBDA}(\mathtt{n}, e)$
Coordinates	$\mathtt{c}$	$::= \mathrm{PARM}(\mathtt{n}, \mathtt{k}) \mid \mathrm{LOCAL}(\mathtt{n}, \mathtt{k})$
Builtins	$\mathtt{r}$	$::= \mathtt{amul} \mid \mathtt{aminus} \mid \ldots$
Integer	$\mathtt{n}$, $\mathtt{k}$	
Identifier	$\mathtt{i}$	
Address	$\mathtt{a}$	

Fig. 3. Poly/ML codetree language

accessed using the PARAM(n, k) codetree term where k denotes the k-th parameter of the function declared n levels out to the current containing function. Tuples are created with the RECORD element and fields of a tuple are extracted with INDIRECT. LAMBDA introduces the body of a function. A BLOCK is a sequence of codetree terms and provides an environment for declarations. The result of evaluating a block is the value of the final expression. Generally, every code tree term except the last will be a DECL codetree term and these have scope over the rest of the block. It is possible to have other kinds of terms within a block that may be executed for their side-effects. Loops can be created using recursive functions or through use of BEGINLOOP and LOOP. BEGINLOOP represents the start of a loop and contains a list of DECL entries that represent loop index variables. The value for each DECL is the initial value of the loop variable. The expression part of the BEGINLOOP will almost always be a nested IF-expression with some of the branches containing LOOP-expressions. A LOOP expression causes a jump back to the BEGINLOOP with the loop variables updated with the values in the LOOP expression. The length of the argument list for a LOOP will always match the containing BEGINLOOP. Branches of the IF-expression that do not end with a LOOP result in exiting the LOOP. There is also no explicit code tree for representing function definition in Poly/ML. A function definition is expressed by a DECL codetree term whose second parameter is a lambda expression. Constants can be expressed as either LIT which represents a literal integer or ADDRESS which represents the address of some entity already present in the ML address space. Poly/ML is an incremental compiler and it is usual to compile an expression which makes reference to pre-existing entities.

As an example of code tree, the following factorial function in ML:

```
fun factorial n =
  if n = 0 then 1
  else n * factorial (n -1);
```

is compiled to the following codetree term:

```
DECL(1,
   LAMBDA(1,
      IF(EVAL(BUILTIN equala, [PARAM(0,1), LIT 0 ]),
         LIT 1,
         BLOCK[DECL(2,
             EVAL(LOCAL(1,1),
                [EVAL(BUILTIN aminus, [PARAM(0,1), LIT 1])])
             ),
             EVAL(BUILTIN amul, [PARAM(0,1), LOCAL(0,2)])
         ]
      )
   )
)
```

The ML function definition is translated to a declaration codetree term which is a DECL expression. The right hand side of the DECL expression is a LAMBDA codetree term representing the body of the function `factorial`. The body contains a conditional, an IF expression whose first argument is the test. If that succeeds the result is

the second argument, the literal value 1, and if it fails the third argument is executed. This consists of a BLOCK. The first entry in the block is a local declaration of a recursive call of `factorial`. The result of this is then multiplied by `n`. Several built-in functions are used. `equala` tests the equality of two integers, and `aminus` does integer subtraction.

5 Generating Poly/ML Codetree from OIL

The task of generating Poly/ML codetree from OIL starts with an OIL module m, a list of top-level declarations. The overall strategy is to translate that cluster of declarations into a Poly/ML codetree term that would eventually evaluate to a record value. Each component of that record is maintained in a one-to-one correspondence with an OIL top-level declaration through an environment of type

$$\text{Env} = [x \mapsto \mathrm{c}].$$

An environment Γ of type Env maps an OIL identifier x to a scope-and-position $\Gamma(x)$, which is either $\texttt{LOCAL}(\mathrm{n}, \mathrm{k})$ for variables or $\texttt{PARAM}(\mathrm{n}, \mathrm{k})$ for function parameters. For top-level declarations, n has value 0. Therefore the most important coordinate information for a top-level declaration is k, which is the slot number for x in the top-level RECORD codetree term.

Notation. In what follows, we use $[\![\bullet]\!]$ to enclose syntactic objects (be they OIL expressions or Poly/ML codetree terms). OIL expression objects are written in *`this font`* whereas Poly/ML codetree terms are typeset with `this other font`. When the meta variable x designates an object from a certain syntactic category, we use the notation $\widehat{\mathrm{x}}$ for a meta variable that holds a sequence of syntactic objects from that same category.

Modules. The entry point to the translation algorithm is the function

$$\mathcal{G} : [\text{Declaration}] \to \text{CodeTree} \times \text{Env}$$

which takes as input a list of OIL top-level declarations and produces a codetree-environment pair. The first component is usually a BLOCK codetree term containing all the codes generated for the top-level declarations and whose last term is a RECORD codetree term that constructs the value representation of the module:

$$\begin{aligned}
&\mathcal{G}(m) = \\
&\quad \textbf{\textit{let}}\ \left\langle \widehat{\mathrm{d}}, \Gamma \right\rangle = \mathcal{D}(m, [], 0, 0) \\
&\qquad \widehat{\mathrm{c}} = \mathcal{C}(\Gamma, 0) \qquad\qquad \text{-- } \textit{reference all toplevel declarations} \\
&\quad \textbf{\textit{in}}\ \left\langle \texttt{BLOCK}(\widehat{\mathrm{d}} + +[\texttt{RECORD}(\widehat{\mathrm{c}})]), \Gamma \right\rangle
\end{aligned}$$

The second component is the resulting environment. The purpose of the the function $\mathcal{D}$

$$\mathcal{D} : [\text{Declaration}] \times \text{Env} \times \text{Integer} \times \text{Integer} \to [\text{CodeTree}] \times \text{Env}$$

is to translate a cluster of declarations in a given initial environment, a scope nesting level, an initial declaration number. It returns a pair of a list of Poly/ML codetree terms for the declarations and an updated environment. The details of code generation for declarations are the subject of the next section. The function $\mathcal{C}$ take an environment Γ, a scope nesting level n, and returns the list of all coordinates in Γ with scope nesting level n.

Definitions. Generating Poly/ML code for an OIL definition is straightforward. We allocate LOCAL coordinates for corresponding Poly/ML entity, and generate codes for the initializer in an environment where the name of the entity being defined is bound to its coordinates

$$
\begin{array}{l}
\mathcal{D}([], \Gamma, n, k) = \langle [], \Gamma \rangle \\
\mathcal{D}\big((\texttt{def}\ x\ e) :: \widehat{d}, \Gamma, n, k\big) = \\
\quad \textbf{\textit{let}}\ \ \Gamma_1 = \Gamma\ ++\ [x \mapsto \texttt{LOCAL}(n,k)] \quad -\!-\ \textit{introduce } x \textit{ in the scope of its initializer} \\
\qquad \langle e, \Gamma_2, k_1 \rangle = \mathcal{E}(e, \Gamma_1, n, k+1) \\
\qquad \left\langle \widehat{d}, \Gamma_3 \right\rangle = \mathcal{D}\left(\widehat{d}, \Gamma_1, n, k_1 + 1\right) \\
\quad \textbf{\textit{in}}\ \left\langle \texttt{DECL}(k, e) :: \widehat{d}, \Gamma_3 \right\rangle
\end{array}
$$

That code generation strategy implements unrestricted value recursion, and in particular recursive functions. If we wanted to restrict recursion to functions only, we could delay to the actual binding the expression translation function $\mathcal{E}$. The function $\mathcal{E}$ with functionality

$$\mathcal{E} : \text{Expression} \times \text{Env} \times \text{Integer} \times \text{Integer} \to \text{CodeTree} \times \text{Env} \times \text{Integer}$$

takes an OIL expression, an initial environment, a scope nesting level, a variable position, and produces a triple that consists of the Poly/ML codetree for the OIL expression, an updated environment, and the next available variable position at the same scope. This function $\mathcal{E}$ (which accepts any OIL expressions) is never called directly to generate codetree for declarations. Its details are the subject of the next paragraphs.

Functional abstraction. Given a functional abstraction of arity k at nesting level n, we augment the enclosing environment G with k PARAM coordinates for the parameters, and then generate code for the body with nesting level increased by one:

$$
\begin{array}{l}
\mathcal{E}([\![(\texttt{lambda}\ (x_1\ \ldots x_k)\ e)]\!], \Gamma, n, k') = \\
\quad \textbf{\textit{let}}\ \ \Gamma_1 = \mathcal{P}(\Gamma, [x_1, \ldots, x_k], n+1) \quad -\!-\ \textit{increase the nesting level for the body} \\
\qquad \langle e, \Gamma_2, k'' \rangle = \mathcal{E}(e, \Gamma_1, n+1, 0) \\
\quad \textbf{\textit{in}}\ \langle \texttt{LAMBDA}(k, e), \Gamma, k' \rangle
\end{array}
$$

Finally we return a LAMBDA codetree term, the original environment and the original variable position number. Allocation of coordinates for the function parameters is done via the helper function

$$\mathcal{P} : \text{Gamma} \times [\text{Identifier}] \times \text{Integer} \to \text{Env}$$

defined by:

$$
\begin{array}{l}
\mathcal{P}(\Gamma, [], n, k) = \Gamma \\
\mathcal{P}(\Gamma, x :: \widehat{x}, n, k) = \Gamma\ ++\ [x \mapsto \texttt{PARAM}(n,k)]\ ++\ \mathcal{P}(\Gamma, \widehat{x}, n, k+1)
\end{array}
$$

Note that translation of `lambda`-expression is the only place where we increase the "nesting level" of codetree coordinates.

Local declarations. Code generation for local declarations is quite similar to that of functional abstraction:

$$
\begin{array}{l}
\mathcal{E}(\llbracket(\texttt{bind}((x_1\ e_1)\ldots(x_k\ e_k))\ e)\rrbracket, \Gamma, n, k') = \\
\quad \textbf{\textit{let}}\ \langle e_1, \Gamma_1, k_1'\rangle = \mathcal{E}(e_1, \Gamma, n, k') \\
\qquad \langle e_2, \Gamma_2, k_2'\rangle = \mathcal{E}(e_2, \Gamma_1 + +[x_1 \mapsto \textsc{local}(n, k_1')], n, k_1' + 1) \\
\qquad \ldots \\
\qquad \langle e_k, \Gamma_k, k_k'\rangle = \mathcal{E}(e_k, \Gamma_{k-1} + +[x_{k-1} \mapsto \textsc{local}(n, k_{k-1}')], n, k_{k-1}' + 1) \\
\qquad \langle e, \Gamma', k''\rangle = \mathcal{E}(e_k, \Gamma_k + +[x_k \mapsto \textsc{local}(n, k_k')], n, k_k' + 1) \\
\quad \textbf{\textit{in}}\ \langle \textsc{block}([\textsc{decl}(k_1', e_1), \ldots, \textsc{decl}(k_k', e_k), e]), \Gamma, k''\rangle
\end{array}
$$

Conditional expressions. The simplest conditional expression is the equivalent of an `if-then` expression without an alternative part:

$$
\begin{array}{l}
\mathcal{E}(\llbracket(\texttt{when}\ ((p_1\ e_1)))\rrbracket, \Gamma, n, k) = \\
\quad \textbf{\textit{let}}\ \langle p_1, \Gamma_1, k_1\rangle = \mathcal{E}(p_1, \Gamma, n, k) \\
\qquad \langle e_1, \Gamma_2, k_2\rangle = \mathcal{E}(e_1, \Gamma_1, n, k_1) \\
\quad \textbf{\textit{in}}\ \langle \textsc{if}(p_1, e_1, \textsc{nil}), \Gamma_2, k_2\rangle
\end{array}
$$

As a special-case, for code-generation purpose, we accept `otherwise` as a predicate with the following meaning:

$$
\begin{array}{l}
\mathcal{E}(\llbracket(\texttt{when}\ ((\texttt{otherwise}\ e)))\rrbracket, \Gamma, n, k) = \\
\quad \mathcal{E}(e, \Gamma, n, k)
\end{array}
$$

The most general form of a conditional expression in OIL is a multiway branch, with an optional default case (the `otherwise`-branch) at the end of list of choices. This form corresponds to a series of nested traditional `if-then-else` expressions:

$$
\begin{array}{l}
\mathcal{E}(\llbracket(\texttt{when}\ (p\ e) :: \widehat{w})\rrbracket, \Gamma, n, k) = \\
\quad \textbf{\textit{let}}\ \langle p, \Gamma_1, k_1\rangle = \mathcal{E}(p, \Gamma, n, k) \\
\qquad \langle e_1, \Gamma_2, k_2\rangle = \mathcal{E}(e, \Gamma_1, n, k_1) \\
\qquad \langle e_2, \Gamma_3, k_3\rangle = \mathcal{E}(\llbracket(\texttt{when}\ \widehat{w})\rrbracket, \Gamma_2, n, k_2) \\
\quad \textbf{\textit{in}}\ \langle \textsc{if}(p_1, e_1, e_2), \Gamma_3, k_3\rangle
\end{array}
$$

Loops. The simplest looping structure in OpenAxiom is the infinite loop represented in OIL as `(loop () e)` where `e` is the expression to be evaluated indefinitely. This basic structure can be controlled by iterators. From a control structure point of view, an iterator is semantically a 4-tuple $\left\langle \widehat{d}, \widehat{e}, p_1, p_2 \right\rangle$ that controls a loop:

1. a sequence $\widehat{d}$ of declarations (and initializations) of variables with lifetime spanning exactly the entire execution of the loop
2. a sequence of expressions $\widehat{e}$ giving values to the variables in the first component for the next attempt at the loop body iteration
3. a filter predicate expression p_1 which controls the evaluation of the body for a particular evaluation of the loop body
4. a continuation predicate p_2 which, if false, terminates the loop

We use the translation function $\mathcal{I}$

$$\mathcal{I} : \text{Control} \times [\text{Iterator}] \times \text{Env} \times \text{Integer} \times \text{Integer} \rightarrow \text{Control} \times \text{Env} \times \text{Integer}$$

as a helper.

$$\mathcal{I}(x, [], \Gamma, n, k) = \langle x, \Gamma, k\rangle$$

$$\mathcal{I}\left(\left\langle \widehat{d}, \widehat{e}, p_1, p_2\right\rangle, [\![(\texttt{while}\ p)]\!] :: \widehat{i}, \Gamma, n, k\right) =$$
$$\textit{let}\ \langle p, \Gamma, k'\rangle = \mathcal{E}(p, \Gamma, n, k)$$
$$p_2' = \text{EVAL}(\text{BUILTIN and}, [p_2, p])$$
$$\textit{in}\ \mathcal{I}\left(\left\langle \widehat{d}, \widehat{e}, p_1, p_2'\right\rangle, \widehat{i}, \Gamma, n, k'\right)$$

$$\mathcal{I}\left(\left\langle \widehat{d}, \widehat{e}, p_1, p_2\right\rangle, [\![(\texttt{until}\ p)]\!] :: i, \Gamma, n, k\right) =$$
$$\textit{let}\ \langle p, \Gamma, k'\rangle = \mathcal{E}(p, \Gamma, n, k)$$
$$p_2' = \text{EVAL}(\text{BUILTIN and}, [p_2, \text{EVAL}\,(\text{BUILTIN not}, [p])])$$
$$\textit{in}\ \mathcal{I}\left(\left\langle \widehat{d}, \widehat{e}, p_1, p_2'\right\rangle, \widehat{i}, \Gamma, n, k'\right)$$

$$\mathcal{I}\left(\left\langle \widehat{d}, \widehat{e}, p_1, p_2\right\rangle, [\![(\texttt{suchthat}\ p)]\!] :: \widehat{i}, \Gamma, n, k\right) =$$
$$\textit{let}\ \langle p, \Gamma, k'\rangle = \mathcal{E}(p, \Gamma, n, k)$$
$$p_1' = \text{EVAL}(\text{BUILTIN and}, [p_1, p])$$
$$\textit{in}\ \mathcal{I}\left(\left\langle \widehat{d}, \widehat{e}, p_1', p_2\right\rangle, \widehat{i}, \Gamma, n, k'\right)$$

$$\mathcal{I}\left(\left\langle \widehat{d}, \widehat{e}, p_1, p_2\right\rangle, [\![(\texttt{step}\ x\ e_1\ e_2\ e_3)]\!] :: \widehat{i}, \Gamma, n, k\right) =$$
$$\textit{let}\ \langle e_1, \Gamma_1, k_1\rangle = \mathcal{E}(e_1, \Gamma, n, k+1) \quad -- \text{k } \textit{is for the loop variable}$$
$$\langle e_2, \Gamma_2, k_2\rangle = \mathcal{E}(e_1, \Gamma_1, n, k_1)$$
$$\langle e_3, \Gamma_3, k_3\rangle = \mathcal{E}(e_2, \Gamma_2, n, k_2+1) \ -- k_2 \textit{ is used for holding the value of } e_2$$
$$p = \text{EVAL}(\text{BUILTIN int\_lss}, [x, \text{LOCAL}\,(n, k_2)])$$
$$e = \text{EVAL}(\text{BUILTIN aplus}, [\text{LOCAL}\,(n, k), \text{LOCAL}\,(n, k_3+1)])$$
$$\Gamma' = \Gamma_3 + +[x \mapsto \text{LOCAL}\,(n, k)]$$
$$\widehat{d}' = [\text{DECL}\,(k, e_1), \text{DECL}\,(k_2, e_2), \text{DECL}\,(k_3+1, e_3)]$$
$$\widehat{e}' = [e, \text{LOCAL}\,(n, k_2), \text{LOCAL}\,(n, k_3+1)]$$
$$x = \left\langle \widehat{d} + +\widehat{d}', \widehat{e} + +\widehat{e}', \text{NIL}, p\right\rangle$$
$$\textit{in}\ \mathcal{I}(x, i, \Gamma', k_3+2)$$

The translation of an OIL loop expression to a Poly/ML codetree term first translates the iterators to control terms, then uses the resulting environment to translate the body of the loop::

$$\mathcal{E}\left([\![(\texttt{loop}\ \widehat{i}\ e)]\!], \Gamma, n, k\right) =$$
$$\textit{let}\ \left\langle\left\langle \widehat{d}, \widehat{e}, p_1, p_2\right\rangle, \Gamma', k'\right\rangle = \mathcal{I}\left(\langle [], \text{NIL}, \text{NIL}, \text{NIL}\rangle, \widehat{i}, \Gamma, n, k\right)$$
$$\langle e, \Gamma'', k''\rangle = \mathcal{E}(e, \Gamma', n, k')$$
$$\textit{in}\ \left\langle \text{BEGINLOOP}\left(\widehat{d}, \text{IF}\,(p_2, \text{BLOCK}\,[\text{IF}\,(p_1, e, \text{NIL}), \text{LOOP}\,\widehat{e}], \text{NIL})\right), \Gamma'', k''\right\rangle$$

Sequence of expressions. A sequence of OIL expressions corresponds to a BLOCK codetree term.

$$\mathcal{E}([\![(\texttt{seq}\ e_1 \ldots e_{n'})]\!], \Gamma, n, k) =$$
$$\textit{let}\ \langle e_1\Gamma_1, k_1\rangle = \mathcal{E}(e_1, \Gamma, n, k)$$

$$
\begin{aligned}
&\ldots\\
&\langle e_{n'}, \Gamma_{n'}, k_{n'}\rangle = \mathcal{E}(e_{n'}, \Gamma_{n'-1}, n, k_{n'-1})\\
\textbf{\textit{in}}\ &\langle \mathrm{BLOCK}\,[e_1, \ldots, e_{n'}], \Gamma_{n'}, k_{n'}\rangle
\end{aligned}
$$

Function calls. An OIL function call expression generally corresponds to an `EVAL` codetree term. The operation may be a builtin operation, a "variable", or a global function (corresponding to a constructor instantiation), or a special runtime function (*e.g.* for `mkCategory` or `Join`) Arguments are evaluated from left to right.

6 Implementation

The code generation component is implemented as an OpenAxiom library. First we compile a Spad program to the OIL intermediate representation. That representation is then translated to Poly/ML codetree terms, written into a file. Note that, the codetree format written to disk is slightly different from the pretty printed version of the Poly/ML compiler. This is done so to make parsing easier, and hide redundant information. One caveat is that the current rule is implemented for category definition without default implementation. Currently the function `Join` only merges specifications from different category objects.

7 Example

Below is the translation of the OIL representation of the category `BasicType`:

```
DECL(1,
 LAMBDA(0,
 BLOCK[
  DECL(2,
   EVAL(ADDRESS Join,
     [
     RECORD[
        EVAL(
          LAMBDA(0,
            BLOCK[
              DECL(2,
                EVAL(ADDRESS Join,
                [
                  RECORD[
                     EVAL(ADDRESS mkCategory, [ADDRESS ?]),
                     LIT 0
                  ]
                ])
              ),
              DECL(3,
                EVAL(ADDRESS setName,
                  [
                    RECORD[LOCAL(0,2), ADDRESS ?]
```

```
                ]
              )
            ),
            LOCAL(0, 3)
          ]
        ),
        [LIT 0]
      ),
      RECORD[EVAL(ADDRESS mkCategory, [ADDRESS ?]), LIT 0]
    ]
    ]
  )
 ),

 DECL(3,
   EVAL(
     ADDRESS setName ,
     RECORD[LOCAL(0,2), ADDRESS ?]
   )
  ),

 LOCAL(0,3)
 ]
)
)
```

8 Related Work

IR based code generations in CAS. The Aldor [21] language compiler defines a first order abstract machine (FOAM) which is an intermediate language for representing Aldor code at a lower level [22]. FOAM is platform independent. However, we have never been able to generate a working FOAM out of AXIOM systems. One of the FOAM's goal is to define data structures for program transformation, optimization at FOAM level, as well as for generating C and Common Lisp code for Aldor programs. Maple [16] uses *inert* expressions to represent a Maple program. The internal representation supports code generation for other languages such as C, Java and Fortran, as well as optimization functionalities provided by Maple's CodeGeneration package. The inert forms closely reflect the Maple internal DAG data structure representation [18].

Interfacing CAS and deduction systems. Various methods for interfacing CAS and deduction systems have been extensively discussed in the literature. In the work of Ballarin and Homann [3], Maple was used as a term rewriting system for enhancing the expression simplifier of Isabelle. Interfaces are provided in Isabelle's ML environment for starting and exiting a Maple session, sending expressions to Maple for evaluation, and receiving results from Maple. The communication at lower level is through a Unix pipe between Maple and Isabelle processes. High-level syntax translation rules are defined to achieve translations between expressions in Isabelle syntax and their equivalents in Maple's syntax. The work of Calmet and Homann [5] as well as the work of

Barendregt and Cohen [4] suggest the use of a separate language such as OpenMath [1] to implement communication protocols between different computation and deduction systems. Harrison and Théry combined HOL and Maple to verify results computed by Maple [11]. The communication is based on the idea of a "software bus". Translation between HOL and Maple terms is implemented as a third party between the two systems. The work of Adams and Dunstan [2] integrated Maple with the automated theorem prover PVS to validate computations in a real analysis library. Maple is extended with external C functions through Maple's foreign function interface. The C functions manage the communications with PVS and high-level syntax translations between different syntax. Recently, Delahaye and Mayero demonstrated a speed-up of the field tactics in Coq with Maple's field computation [10]. Algebraic expressions over a field in Coq are translated to the expressions in Maple's syntax, and sent to Maple for computation. The results are sent back to Coq and translated back into Coq's syntax.

Our approach differs from all the approaches mentioned above in several aspects. The existing approaches rely mainly on defining: (1) process communication protocols between CAS and deduction systems and (2) high-level program syntax translations. This results in the CAS and the deduction system each having each their own address space. Our work generates the same lower level run time instructions from programs with different syntax. OpenAxiom programs and ML programs will share the same address space. Translation rules in our work are defined between intermediate representations instead of high-level syntax. Furthermore, improvements to the Poly/ML system (such as recent addition of efficient and scalable concurrency primitives) are directly available to both OpenAxiom and Poly/ML-based engines. We consider this aspect to be of a significant benefit. It will also help enhance and improve recent implicit parallelization capabilities [14,15] added to OpenAxiom.

9 Conclusion and Future Work

This paper is a report on a work-in-progress. Obviously, much remains to be done for runtime domain instantiations. In the long-term, we would like to obtain a complete formal specification of the Spad programming language, including its intermediate representation. Ideally, we would like a formally checked translation. But, that is a much harder task that will span several years. In the near future, after the complete re-targeting of OpenAxiom, we plan to investigate various connections between several Poly/ML based logical systems (Isabelle in particular) and the OpenAxiom platform as completion of our initial goal.

Acknowledgements. This work was partially supported by NSF grant CCF-1035058.

References

1. Abbott, J., Díaz, A., Sutor, R.S.: A report on openmath: a protocol for the exchange of mathematical information. SIGSAM Bull. 30, 21–24 (1996)
2. Adams, A., Dunstan, M., Gottliebsen, H., Kelsey, T., Martin, U., Owre, S.: Computer algebra meets automated theorem proving: Integrating maple and pvs. In: Boulton, R.J., Jackson, P.B. (eds.) TPHOLs 2001. LNCS, vol. 2152, pp. 27–42. Springer, Heidelberg (2001)

3. Ballarin, C., Homann, K., Calmet, J.: Theorems and algorithms: an interface between isabelle and maple. In: Proceedings of the 1995 international symposium on Symbolic and algebraic computation, ISSAC 1995, pp. 150–157. ACM, New York (1995)
4. Barendregt, H., Cohen, A.M.: Electronic communication of mathematics and the interaction of computer algebra systems and proof assistants. J. Symb. Comput. 32, 3–22 (2001)
5. Calmet, J., Homann, K.: Classification of communication and cooperation mechanisms for logical and symbolic computation systems. In: Frontiers of Combining Systems (FroCos), pp. 221–234 (1996)
6. Carette, J., Farmer, W.M., Wajs, J.: Trustable communication between mathematics systems. In: Proc. of Calculemus 2003, Aracne, pp. 58–68 (2003)
7. Crary, K., Weirich, S., Morrisett, G.: Intensional Polymorphism in Type-erasure Semantics. J. Funct. Program. 12, 567–600 (2002)
8. Davenport, J.H.: A New Algebra System. Technical report, IBM Research (1984)
9. Davenport, J.H.: Equality in Computer Algebra and Beyond. J. Symb. Comput. 34, 259–270 (2002)
10. Delahaye, D., Mayero, M.: Dealing with algebraic expressions over a field in coq using maple. J. Symb. Comput. 39, 569–592 (2005)
11. Harrison, J., Théry, L.: A skeptic's approach to combining hol and maple. J. Autom. Reason. 21, 279–294 (1998)
12. Isabelle (2011), http://www.cl.cam.ac.uk/research/hvg/Isabelle/index.html
13. Jenks, R.D., Sutor, R.S.: AXIOM: The Scientific Computation System. Springer, Heidelberg (1992)
14. Li, Y., Reis, G.D.: A Quantitative Study of Reductions in Algebraic Libraries. In: PASCO 2010: Proceedings of the 4th International Workshop on Parallel and Symbolic Computation, pp. 98–104. ACM, New York (2010)
15. Li, Y., Reis, G.D.: An Automatic Parallelization Framework for Algebraic Computation Systems. In: ISSAC 2011: Proceedings of the International Symposium on Symbolic and Algebraic Computation. ACM, New York (2011)
16. Maple. Maplesoft Inc., Canada (2011), http://www.maplesoft.com
17. Matthews, D.C.J., Wenzel, M.: Efficient Parallel Programming in Poly/ML and Isabelle/ML. In: Proceedings of the 5th ACM SIGPLAN workshop on Declarative aspects of multicore programming, DAMP 2010, pp. 53–62. ACM, New York (2010)
18. Monagan, M.B., Geddes, K.O., Heal, K.M., Labahn, G., Vorkoetter, S.M., McCarron, J., De Marco, P.: Maple Advanced Programming Guide, Maplesoft, Canada (2007)
19. OpenAxiom (2011), http://www.open-axiom.org
20. Poly/ML (2011), http://www.polyml.org
21. Watt, S.M.: Aldor Programming Language (December 2009), http://www.aldor.org
22. Watt, S.M., Broadbery, P.A., Iglio, P., Morrison, S.C., Steinbach, J.M.: Foam: A first order abstract machine version 0.35. Technical report, IBM Thomas J. Watson Research Center (2001)

Incidence Simplicial Matrices Formalized in Coq/SSReflect*

Jónathan Heras[1], María Poza[1], Maxime Dénès[2], and Laurence Rideau[2]

[1] Department of Mathematics and Computer Science of University of La Rioja
[2] INRIA Sophia Antipolis - Méditerranée
{jonathan.heras,maria.poza}@unirioja.es,
{Maxime.Denes,Laurence.Rideau}@inria.fr

Abstract. Simplicial complexes are at the heart of Computational Algebraic Topology, since they give a concrete, combinatorial description of otherwise rather abstract objects which makes many important topological computations possible. The whole theory has many applications such as coding theory, robotics or digital image analysis. In this paper we present a formalization in the Coq theorem prover of simplicial complexes and their incidence matrices as well as the main theorem that gives meaning to the definition of homology groups and is a first step towards their computation.

1 Introduction

Algebraic Topology is a vast and complex subject, in particular mixing Algebra and (combinatorial) Topology. Algebraic Topology consists of trying to use as much as possible "algebraic" methods to attack topological problems. For instance, one can define some special groups associated with a topological space, in a way that respects the relation of homeomorphism of spaces. This allows one to study properties about topological spaces by means of statements about groups, which are often easier to prove.

However, in spite of being an abstract mathematical subject, Algebraic Topology methods can be implemented in software systems and then applied to different contexts such as coding theory [23], robotics [17] or digital image analysis [13,14] (in this last case, in particular in the study of medical images [21]). Nevertheless, if we want to use these systems in real life problems, we have to be completely sure that the systems are correct. Therefore, to increase the reliability of these methods and the systems that implement them, we can use Theorem Proving tools. In this paper we are going to focus on the verification of some results about a mathematical structure which can be useful, among others things, to study properties of digital images.

Simplicial complexes are topological abstract structures which provide a good framework to apply topological methods to analyse digital images. Intuitively, a

* Partially supported by Ministerio de Educación y Ciencia, project MTM2009-13842-C02-01, and by European Community FP7, STREP project ForMath.

J.H. Davenport et al. (Eds.): Calculemus/MKM 2011, LNAI 6824, pp. 30–44, 2011.

simplicial complex is a generalization of the notion of graph to higher dimensions. Indeed, all the simplicial complexes of dimension less than two are graphs.

A central problem in this context consists of computing homology groups of simplicial complexes. Homology groups characterize both the number and the type of holes and the number of connected components of a simplicial complex. This type of information is used, for instance, to determine similarities between proteins in molecular biology [7].

In the context of the computation of homology groups, we can highlight the Kenzo program [9], a successful Computer Algebra system, implemented in Common Lisp, which has obtained some homology groups not confirmed nor refuted by any other means.

There are two different ways of computing homology groups in Kenzo depending on the type of the object. On the one hand, the task of calculating homology groups of a *finite object* is translated to a problem of diagonalizing certain matrices called *incidence matrices*, see [22]. On the other hand, in the case of *non-finite type* objects, Sergeraert's effective homology [20] theory, implemented in Kenzo, provides a framework where this question can be handled. Roughly speaking, the effective homology method links a non-finite type object, X, with a finite type object, Y, with the same homology groups; then the problem of computing the homology groups of X is reduced to the task of diagonalizing the *incidence matrices* of Y.

Sergeraert's ideas have been translated to theorem provers with the aim of not only formalizing the effective homology theory, but also applying formal methods to the study of Kenzo. Thus far, the main formalization efforts have been focused on theorems which provide the connection between non-finite type objects with finite type ones; here, we can distinguish the verification of the Basic Perturbation Lemma in the Isabelle/HOL proof assistant, see [2], or the formalization in Coq of the Effective Homology of Bicomplexes, see [8].

However, up to now, the question of formalizing the computation of homology groups of finite objects has not been undertaken. In this paper we discuss the formalization of simplicial complexes and their incidence matrices as well as the main theorem that gives meaning to the definition of homology groups. To this aim, we have used the proof assistant Coq [6,4] as well as the SSReflect extension [11] and the libraries it provides.

The rest of the paper is organized as follows. Section 2 contains some preliminaries on Algebraic Topology. A sketch of the proof of the main theorem is presented in Section 3. A brief introduction to SSReflect is provided in Section 4. The main steps of the formalization are given in Section 5. The paper ends with a section of Conclusions and Further Work, and the bibliography.

2 Mathematical Preliminaries

In this section, we briefly provide the minimal standard background needed in the rest of the paper. We mainly focus on definitions. Many good textbooks are available for these definitions and results about them, the main one being maybe [18].

The notion of simplicial complex gives rise to the most elementary method to settle a connection between common Topology and Algebraic Topology. The notion of topological space is too *abstract* to perform computations. Simplicial complexes provide a purely combinatorial description of topological spaces which admit a triangulation. The computability of properties, such as homology groups, from a simplicial complex associated with a topological space is well-known and the algorithm uses simple linear algebra [22]. Then, an algebraic topologist can decide every sensible space (that is to say, a topological space which admit a triangulation) is a simplicial complex, making computations easier.

Let us start with some basic terminology. Let V be an ordered set, called the *vertex set*. An *(abstract) simplex* over V is any finite subset of V. An *(abstract) n-simplex* over V is a simplex over V whose cardinality is equal to $n+1$. Given a simplex α over V, we call subsets of α *faces* of α.

Definition 1. *An* (ordered abstract) simplicial complex *over V is a set of simplices $\mathcal{K}$ over V such that it is closed by taking faces (subsets); that is to say, if $\alpha \in \mathcal{K}$ all the faces of α are in $\mathcal{K}$, too.*

Let $\mathcal{K}$ be a simplicial complex. Then the set $S_n(\mathcal{K})$ of n-simplices of $\mathcal{K}$ is the set made of the simplices of cardinality $n+1$.

Example 1. Let us consider $V = (0, 1, 2, 3, 4, 5, 6)$.

The small simplicial complex drawn in Figure 1 is mathematically defined as the object:

$$\mathcal{K} = \left\{ \begin{array}{l} \emptyset, (0), (1), (2), (3), (4), (5), (6), (0,1), (0,2), (0,3), (1,2), \\ (1,3), (2,3), (3,4), (4,5), (4,6), (5,6), (0,1,2), (4,5,6) \end{array} \right\}$$

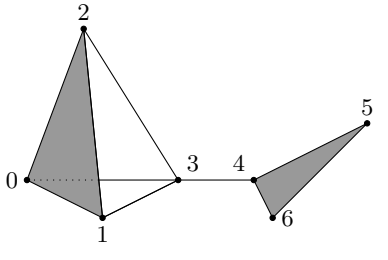


Fig. 1. Butterfly Simplicial Complex

It is worth noting that simplicial complexes can be infinite. For instance if $V = \mathbb{N}$ and the simplicial complex $\mathcal{K}$ is $\{(n)\}_{n\in\mathbb{N}} \cup \{(0, n)\}_{n\geq 1}$, the simplicial complex obtained can be seen as an infinite bunch of segments.

Definition 2. *A facet of a simplicial complex $\mathcal{K}$ over V is a maximal simplex with respect to the subset order $\subseteq$ among the simplexes of $\mathcal{K}$.*

Example 2. The facets of the simplicial complex depicted in Figure 1 are:

$$\{(0,3), (1,3), (2,3), (3,4), (0,1,2), (4,5,6)\}$$

To construct the simplicial complex associated with a sequence of facets, $\mathcal{F}$, we generate all the faces of the simplexes of $\mathcal{F}$. Subsequently, if we perform the set union of all the faces we obtain the simplicial complex associated with $\mathcal{F}$.

Definition 3. *Let $\mathcal{K}$ be a simplicial complex over V. Let n and i be two integers such that $n \geq 1$ and $0 \leq i \leq n$. Then the* face operator ∂_i^n *is the linear map $\partial_i^n : S_n(\mathcal{K}) \to S_{n-1}(\mathcal{K})$ defined by:*

$$\partial_i^n((v_0, \ldots, v_n)) = (v_0, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n)$$

the i-th vertex of the simplex is removed, so that an $(n-1)$-simplex is obtained.

Now, we are going to introduce a central notion in Algebraic Topology. We assume as known the notions of ring, module over a ring and module morphism (see [16] for details).

Definition 4. *Given a ring R, a* graded module *M is a family of left R-modules $(M_n)_{n\in\mathbb{Z}}$.*

Definition 5. *Given a pair of graded modules M and M', a* graded module morphism *f of degree k between them is a family of module morphisms $(f_n)_{n\in\mathbb{Z}}$ such that $f_n : M_n \to M'_{n+k}$ for all $n \in \mathbb{Z}$.*

Definition 6. *Given a graded module M, a* differential *$(d_n)_{n\in\mathbb{Z}}$ is a family of module endomorphisms of M of degree -1 such that $d_{n-1} \circ d_n = 0$ for all $n \in \mathbb{Z}$.*

The previous definitions define a graded structure and a way of going from a level of the structure to the inferior one. From the previous definitions, the notion of chain complex is defined as follows.

Definition 7. *A* chain complex *C_* is a family of pairs $(C_n, d_n)_{n\in\mathbb{Z}}$ where $(C_n)_{n\in\mathbb{Z}}$ is a graded module and $(d_n)_{n\in\mathbb{Z}}$ is a differential on $(C_n)_{n\in\mathbb{Z}}$.*

The module C_n is called the module of n-chains. *The image $B_n = im\ d_{n+1} \subseteq C_n$ is the (sub)module of n-boundaries. The kernel $Z_n = ker\ d_n \subseteq C_n$ is the (sub)module of* n-cycles.

Given a chain complex $C_* = (C_n, d_n)_{n\in\mathbb{Z}}$, the identities $d_{n-1} \circ d_n = 0$ are equivalent to the inclusion relations $B_n \subseteq Z_n$: every boundary is a cycle but the converse is not generally true. Thus, the next definition makes sense.

Definition 8. *Let $C_* = (C_n, d_n)_{n\in\mathbb{Z}}$ be a chain complex of R-modules. For each degree $n \in \mathbb{Z}$, the* n-homology module *of C_* is defined as the quotient module*

$$H_n(C_*) = \frac{Z_n}{B_n}$$

Once we have defined the notions of simplicial complexes and chain complexes, we can define the link between them considering $\mathbb{Z}$ as the ring R; the most common case in Algebraic Topology.

Definition 9. *Let $\mathcal{K}$ be a simplicial complex over V. Then* the chain complex $C_*(\mathcal{K})$ canonically associated with $\mathcal{K}$ *is defined as follows. The chain group $C_n(\mathcal{K})$ is the free $\mathbb{Z}$ module generated by the n-simplices of $\mathcal{K}$. In addition, let $(v_0, \ldots, v_n)$ be an n-simplex of $\mathcal{K}$, the differential of this simplex is defined as:*

$$d_n := \sum_{i=0}^{n} (-1)^i \partial_i^n$$

In order to clarify the notion of chain complex canonically associated with a simplicial complex, let us present an example. The chain complexes associated with simplicial complexes are good candidates for this purpose.

Example 3. Let $\mathcal{K}$ be the simplicial complex defined in Figure 1. The chain complex $C_*(\mathcal{K})$ canonically associated with $\mathcal{K}$ is:

$$\cdots \to 0 \to C_2(\mathcal{K}) \xrightarrow{d_2} C_1(\mathcal{K}) \xrightarrow{d_1} C_0(\mathcal{K}) \to 0 \to \cdots$$

where there are 3 associated chain groups:

- $C_0(\mathcal{K})$, the free $\mathbb{Z}$-module on the set of 0-simplexes (vertices) $\{(0), (1), (2), (3), (4), (5), (6)\}$.
- $C_1(\mathcal{K})$, the free $\mathbb{Z}$-module on the set of 1-simplexes (edges) $\{(0,1), (0,2), (0,3), (1,2), (1,3), (2,3), (3,4), (4,5), (4,6), (5,6)\}$.
- $C_2(\mathcal{K})$, the free $\mathbb{Z}$-module on the set of 2-simplexes (triangles) $\{(0,1,2), (4,5,6)\}$.

The elements of either of those groups C_p are linear integer combinations of the corresponding basis (set of σ_i's), i.e. elements of the form $\sum \lambda_i \sigma_i$, $\lambda_i \in \mathbb{Z}$.

The differential homomorphism is in this case:

$$d_n((v_0, \ldots, v_n)) := \sum_{i=0}^{n} (-1)^i (v_0, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n) \tag{1}$$

For instance, $d_2((0,1,2)) = (1,2) - (0,2) + (0,1)$.

From the previous definition, we can introduce a very useful concept for the computation of homology groups of simplicial complexes.

Definition 10. *Let $\mathcal{K}$ be a simplicial complex over V and let n be an integer such that $n \geq 1$. The* n-th incidence matrix of $\mathcal{K}$ *over the ring $\mathbb{Z}$, denoted by $M_n(\mathcal{K}, \mathbb{Z})$, represents the $(n-1)$-simplices of $\mathcal{K}$ as rows and the n-simplices of $\mathcal{K}$ as columns. Assuming an ordering on the simplices of the same dimension (in the rest of the paper we assume that the simplices of the same dimension will be ordered), $M_n(\mathcal{K}, \mathbb{Z})$ is $[a_i^j]$ where i ranges from 1 to the cardinality of $S_{n-1}(\mathcal{K})$, j ranges from 1 to the cardinality of $S_n(\mathcal{K})$ and the value of a_i^j is the coefficient of the i-th $(n-1)$-simplex in the differential of the j-th n-simplex; then a_i^j is a value in $\{0, \pm 1\}$.*

Example 4. If we impose a lexicographical order on the simplices of the same dimension of the simplicial complex depicted in Figure 1 (if $v = (a_0, \ldots, a_n)$ and $w = (b_0, \ldots, b_n)$ are n-simplices of the simplicial complex, then $v < w$ if $a_0 < b_0$, or $a_0 = b_0$ and $a_1 < b_1$, or $a_0 = b_0$ and $a_1 = b_1$ and $a_2 < b_2$,..., or $a_0 = b_0, \ldots a_{n-1} = b_{n-1}$ and $a_n < b_n$), then its first incidence matrix is:

$$
\begin{array}{c}
 \\
(0) \\
(1) \\
(2) \\
(3) \\
(4) \\
(5) \\
(6)
\end{array}
\begin{array}{c}
\begin{array}{cccccccccc}
(0,1) & (0,2) & (0,3) & (1,2) & (1,3) & (2,3) & (3,4) & (4,5) & (4,6) & (5,6)
\end{array} \\
\left(\begin{array}{cccccccccc}
-1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 1 & -1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1
\end{array}\right)
\end{array}
$$

The relevance of the incidence matrices of simplicial complexes lies in the fact that they can be used to compute the homology groups of the simplicial complex by means of a diagonalization process, as explained for instance in [22].

3 The Theorem Formalized and Its Context

The definitions presented in the previous section are classical definitions from Algebraic Topology. However, since our final goal consists of working with mathematical objects coming from digital images, let us show how this machinery from algebraic topology may be used in this context.

It is worth noting that there are several methods to construct a simplicial complex from a digital image [3]. We are going to explain one of these methods. Roughly speaking, the chosen method consists of obtaining a sequence of facets from a digital image. Then, as we have explained in the previous section, we can obtain the simplicial complex associated with the facets. So, we only need to explain how to get the facets from a digital image.

We are going to work with monochromatic two dimensional images. An image can be represented by a finite 2-dimensional array of 1's and 0's in which the black pixels are represented by 1's and white pixels are represented by 0's.

Let $\mathcal{I}$ be an image codified as a 2-dimensional array of 1's and 0's. Let $V = (\mathbb{N}, \mathbb{N})$ be the vertex set, each vertex is a pair of natural numbers. Let $p = (a, b)$ be the coordinates of a black pixel in $\mathcal{I}$. For each p we can obtain two 2-simplexes which are two facets of the simplicial complex associated with $\mathcal{I}$. Namely, for each $p = (a, b)$ we obtain the following facets: $((a, b), (a+1, b), (a+1, b+1))$ and $((a, b), (a, b+1), (a+1, b+1))$. If we repeat the process for the coordinates of all the black pixels in $\mathcal{I}$, we obtain the facets of a simplicial complex associated with $\mathcal{I}$, let us called it $\mathcal{K}_{\mathcal{I}}$.

Example 5. Consider the image depicted in Figure 2. This image, $\mathcal{I}$, can be codified by means of the 2-dimensional array: ((1,0),(0,1)). Then, with the previously explained process we obtain the facets of $\mathcal{K}_{\mathcal{I}}$. The coordinates of the black pixels are $(0, 0)$ and $(1, 1)$, so the facets that we obtain are:

$$(((0,0),(1,0),(1,1)),((0,0),(0,1),(1,1)),((1,1),(2,1),(2,2)),((1,1),(1,2),(2,2)))$$

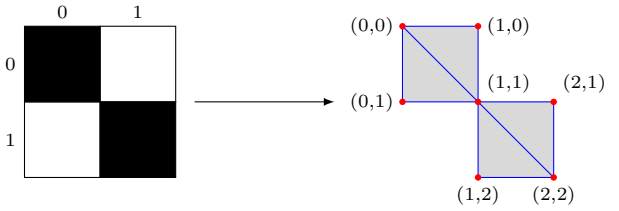


Fig. 2. A digital image and its simplicial complex representation

We have presented a method to obtain a simplicial complex associated with a 2D-image, this process can be generalized to higher-dimensional images [19].

It is worth noting that even the bigger digital images have always a finite number of components, hence a finite number of vertices and then our vertex set V consists of a finite number of vertices. Therefore, the simplicial complexes coming from digital images are always of finite type. This point will be important in our formalization.

Moreover, instead of working with the ring $\mathbb{Z}$, we consider the ring $\mathbb{Z}/2\mathbb{Z}$ since the computation of homology groups is easier working with $\mathbb{Z}/2\mathbb{Z}$. This approach is usually followed when algebraic topology methods are applied to the study of digital images, see [13,14].

Then, we are going to work with a different definition of the face operator and associated incidence matrices. Indeed, since coefficients (in $\mathbb{Z}/2\mathbb{Z}$) of opposite sign are identified, we do not have to deal with orientations of faces.

Thus, in the following $\mathcal{K}$ will denote a simplicial complex over a finite set V and n an integer such that $n \geq 1$. The incidence matrix is now defined by:

Definition 11. *The* n-th incidence matrix of $\mathcal{K}$ *over the ring* $\mathbb{Z}/2\mathbb{Z}$*, denoted by* $M_n(\mathcal{K})$*, is a matrix of size* $m \times p$*, where* m *is the cardinality of* $S_{n-1}(\mathcal{K})$ *and* p *is cardinality of* $S_n(\mathcal{K})$*. Its coefficients* $[a_i^j]$ *are* 1 *if the* i*-th* $(n-1)$*-simplex is a face of the* j*-th* n*-simplex and* 0 *otherwise.*

Note that the *n-th incidence matrix of $\mathcal{K}$* over the ring $\mathbb{Z}/2\mathbb{Z}$ is the absolute value of the *n-th incidence matrix of $\mathcal{K}$* over the ring $\mathbb{Z}$.

Using this definition of incidence matrices, it is not necessary to use chain complexes to compute homology groups of simplicial complexes, but just applying a diagonalization process, as described in [22].

Example 6. If we impose a lexicographical order on the simplices of the same dimension of the simplicial complex depicted in Figure 1, then its first incidence matrix over the ring $\mathbb{Z}/2\mathbb{Z}$ is:

$$
\begin{array}{c}
 \\
(0) \\
(1) \\
(2) \\
(3) \\
(4) \\
(5) \\
(6)
\end{array}
\begin{array}{c}
\begin{array}{cccccccccc}
(0,1) & (0,2) & (0,3) & (1,2) & (1,3) & (2,3) & (3,4) & (4,5) & (4,6) & (5,6)
\end{array} \\
\left(\begin{array}{cccccccccc}
1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1
\end{array}\right)
\end{array}
$$

As we have said previously, incidence matrices of simplicial complexes come from the differentials of the chain complexes canonically associated with the simplicial complexes. Theses differentials satisfy a nilpotency condition ($d_{n-1} \circ d_n = 0$).

Then, we can state and proof the following theorem that is analogous to this nilpotency condition on the incidence matrices we have defined above. It should be noted that the statement below is the immediate transcription of the one we formalized and proved in Coq/SSReflect.

Theorem 1. *The product of the n-th incidence matrix of $\mathcal{K}$ over the ring $\mathbb{Z}/2\mathbb{Z}$, $M_n(\mathcal{K})$, and the $(n+1)$-incidence matrix of $\mathcal{K}$ over the ring $\mathbb{Z}/2\mathbb{Z}$, $M_{n+1}(\mathcal{K})$ is equal to the null matrix.*

Sketch of the proof. Let S_{n-1}, S_n, S_{n+1} be the set of $(n-1)$-simplices of $\mathcal{K}$, the set of n-simplices of $\mathcal{K}$ and the set of $(n+1)$-simplices of $\mathcal{K}$ respectively. Then,

$$M_n(\mathcal{K}) = \begin{array}{c} \\ S_{n-1}[1] \\ \vdots \\ S_{n-1}[r2] \end{array} \begin{array}{c} \begin{array}{ccc} S_n[1] & \cdots & S_n[r1] \end{array} \\ \begin{pmatrix} a_{1,1} & \cdots & a_{1,r1} \\ \vdots & \ddots & \vdots \\ a_{r2,1} & \cdots & a_{r2,r1} \end{pmatrix} \end{array}, M_{n+1}(\mathcal{K}) = \begin{array}{c} \\ S_n[1] \\ \vdots \\ S_n[r1] \end{array} \begin{array}{c} \begin{array}{ccc} S_{n+1}[1] & \cdots & S_{n+1}[r3] \end{array} \\ \begin{pmatrix} b_{1,1} & \cdots & b_{1,r1} \\ \vdots & \ddots & \vdots \\ b_{r1,1} & \cdots & b_{r1,r3} \end{pmatrix} \end{array}$$

where $r1 = \sharp|S_n|$, $r2 = \sharp|S_{n-1}|$ and $r3 = \sharp|S_{n+1}|$. Thus,

$$M_n(\mathcal{K}) \times M_{n+1}(\mathcal{K}) = \begin{pmatrix} c_{1,1} & \cdots & c_{1,r3} \\ \vdots & \ddots & \vdots \\ c_{r2,1} & \cdots & c_{r2,r3} \end{pmatrix} \quad \text{where } c_{i,j} = \sum_{1 \le k \le r1} a_{i,k} \times b_{k,j}$$

To prove that $M_n \times M_{n+1}$ is equal to the null matrix, it is enough to prove that $\forall i, j$ such that $1 \le i \le \sharp|S_{n-1}|$ and $1 \le j \le \sharp|S_{n+1}|$, then $c_{i,j} = 0$. Each of these coefficients is written:

$$c_{i,j} = \sum_{1 \le k \le r1} a_{i,k} \times b_{k,j}$$

Since k enumerates the indices of elements of S_n, we may write:

$$c_{i,j} = \sum_{X \in S_n} F(S_{n-1}[i], X) \times F(X, S_{n+1}[j]) \quad \text{with } F(Y, Z) = \begin{cases} 1 \text{ if } Y \in dZ \\ 0 \text{ otherwise} \end{cases} \quad (2)$$

dZ is the analogous in our context of the differential operator defined by (1) and is equal to:

$$dZ = \{Z \setminus \{x\} \mid x \in Z\}$$

This summation can be split depending on whether $X \in \partial S_{n+1}[j]$ or $X \notin \partial S_{n+1}[j]$.

$$c_{i,j} = \sum_{X \in S_n | X \in \partial S_{n+1}[j]} F(S_{n-1}[i], X) \times 1 \tag{3}$$
$$+ \sum_{X \in S_n | X \notin \partial S_{n+1}[j]} F(S_{n-1}[i], X) \times 0$$
$$= \sum_{X \in S_n | X \in \partial S_{n+1}[j]} F(S_{n-1}[i], X) \tag{4}$$

The last summation is expressed over the image of the face operator $x \mapsto S_{n+1}[j] \setminus \{x\}$ which, restricted to $S_{n+1}[j]$, is injective. Thus, we can reindex:

$$c_{i,j} = \sum_{x \in S_{n+1}[j]} F(S_{n-1}[i], S_{n+1}[j] \setminus \{x\}) \tag{5}$$

Subsequently, this summation can also be split depending on whether $x \in S_{n-1}[i]$ or $x \notin S_{n-1}[i]$.

$$c_{i,j} = \sum_{x \in S_{n+1}[j] | x \in S_{n-1}[i]} F(S_{n-1}[i], S_{n+1}[j] \setminus \{x\}) +$$
$$\sum_{x \in S_{n+1}[j] | x \notin S_{n-1}[i]} F(S_{n-1}[i], S_{n+1}[j] \setminus \{x\}) \tag{6}$$

Let us note that if $x \in S_{n-1}[i]$ then $S_{n-1}[i] \not\subset S_{n+1}[j] \setminus \{x\}$, hence the first sum above is 0.

$$c_{i,j} = \sum_{x \in S_{n+1}[j] | x \notin S_{n-1}[i]} F(S_{n-1}[i], S_{n+1}[j] \setminus \{x\}) \tag{7}$$

Here, we can split our proof considering two cases: $S_{n-1}[i] \not\subset S_{n+1}[j]$ and $S_{n-1}[i] \subset S_{n+1}[j]$.

In the first case, we have: $\forall x \in S_{n-1}[i], F(S_{n-1}[i], S_{n+1}[j] \setminus \{x\}) = 0$, hence the result holds.

In the second case, $S_{n-1}[i] \subset S_{n+1}[j]$ implies that if $x \notin S_{n-1}[i]$ then $S_{n-1}[i] \in \partial S_{n+1}[j] \setminus \{x\}$, so the terms are all 1.

$$c_{i,j} = \sum_{x \in S_{n+1}[j] | x \notin S_{n-1}[i]} 1 \tag{8}$$
$$= \sharp |S_{n+1}[j] \setminus S_{n-1}[i]|$$
$$= n + 2 - n = 2 = 0 \bmod 2$$

4 SSReflect Basics

To formalize Theorem 1, we have used SSReflect [11], an extension for the Coq proof assistant [4,6]. Its development was started by G. Gonthier during

the formal proof of the Four Color Theorem [10] and is now involved in the formalisation of the Feit-Thompson theorem [1].

SSReflect (for Small Scale Reflection) introduces a new language for tactics that eases the development of proof scripts. Another main feature is the generic reflection mechanism. More details on the SSReflect tactics language and reflection techniques are presented in its manual [11].

Moreover, SSReflect provides a set of libraries embedding definitions and properties for a variety of mathematical structures. In our formalization, it is worth mentioning the following libraries:

- matrix.v: this library formalizes matrix theory, determinant theory and matrix decompositions. In our problem, this library is used to define incidence matrices, and to state and prove Theorem 1.
- finset.v and fintype.v: theory of finite sets and finite types. We use these libraries to define the basic concepts about simplicial complexes.
- bigop.v: generic indexed "big" operations, like $\sum_{i=0}^{n} f(i)$ or $\bigcup_{i \in I} f(i)$ and their properties, which are useful to deal with the matricial product in Theorem 1.
- zmodp.v: additive group and ring $\mathbb{Z}_p$, together with field properties when p is a prime. As we work with elements of the field $\mathbb{F}_2$, we need this library.

For more precise details on these libraries we refer to [5,12]

5 Formal Development

The SSReflect libraries include all the necessary ingredients to represent the mathematical structures of our formalization.

First of all, we define the notions related to simplicial complexes. The vertices are represented by a finite type `V`. A simplex is defined as a finite set of vertices. Then, the definition of a simplicial complex as a set of simplices closed under inclusion is straightforward:

```
Variable V : finType.
Definition simplex := {set V}.
Definition simplicial_complex (c : {set simplex}) :=
  forall x, x \in c -> forall y : simplex, y \subset x -> y \in c.
```

Since we do not take into account the signs of the coefficients appearing in the incidence matrices, we define a face operator as a set difference (we remove a vertex from a simplex) and the boundary as the image of a simplex by the face operator.

```
Definition face_op (S : simplex) (x : V) := S :\ x.
Definition boundary (S : simplex) := (face_op S) @: S.
```

We prove the correctness of our definition of boundary by showing it is equivalent to a subset relation with constraints on cardinality:

```
Lemma boundaryP: forall (S : simplex) (B : simplex),
  reflect (B \subset S /\ #|S| = #|B|.+1) (B \in boundary S).
```

The statement `reflect P b` expresses an equivalence between a proposition `P` and a boolean expression `b`. This allows to take advantage of the decidability of some propositions by going back and forth from their logical expressions (useful for reasoning) to their boolean counterparts (well suited for computations).

A key argument for our proof is the injectivity of the face operator above, which we establish as a lemma:

```
Lemma face_op_inj2: forall (S : simplex),
  {in S &, injective (face_op S)}.
```

The notation `{in S &, P}` performs localization of predicates: if `P` is of the form `forall x y, Qxy` then `{in S &, P}` means `forall x y, x \in S -> y \in S -> Qxy`. In our case, `injective f` stands for `forall x y, f x = f y -> x = y`.

Now, before giving the definition of the n-th incidence matrix of a simplicial complex, we can define the more generic notion of incidence matrix of two finite sets of simplices.

Representing a matrix requires an indexing of the simplices in `Left` (for the rows) and `Top` (for the columns). Since `Left` and `Top` are finite sets, they are equipped with a canonical enumeration: `(enum_val Left i)` returns the i-th element of the set `Left`. A coefficient a_{ij} of the incidence matrix will be 1 if the i-th simplex of `Left` is a face (subset) of the j-th simplex of `Top` and 0 otherwise.

Thus we can define the incidence matrix of two finite sets of simplices as follows:

```
Variables Left Top : {set simplex}.
Definition incidenceMatrix :=
  \matrix_(i < #|Left|, j < #|Top|)
      if enum_val i \in boundary (enum_val j) then 1 else 0:'F_2.
```

In the definition above, it can be noted that the first argument of `enum_val` is implicit and determined by the context. Indeed, the notation `i < #|Left|` means that the type of `i` is `'I_(#|Left|)`, that is `i` is an ordinal ranging from 0 to `#|Left|`-1, where `#|X|` denotes the cardinal of the set `X`. With this type information, the system expands `enum_val i` to `enum_val Left i`, thus resolving the ambiguity (and similarly for `j`).

The type annotation `0:'F_2` indicates that the 0 and 1 appearing as coefficients of the matrix are the two elements of $\mathbb{F}_2$, that is $\mathbb{Z}/2\mathbb{Z}$ as a field.

We now define the `n`-th incidence matrix of a simplicial complex `c`, by instantiating Left to the set of $n-1$-simplices (of c) and Top to the set of n-simplices. Note that n should be nonzero.

```
Section nth_incidence_matrix.
Variable c: {set simplex}.
Variable n:nat.
Definition n_1_simplices := [set x \in c | #|x| == n].
```

```
Definition n_simplices := [set x \in c | #|x| == n+1].
Definition incidence_matrix_n :=
  incidenceMatrix n_1_simplices n_simplices.
End nth_incidence_matrix.
```

Then we have all the ingredients to state Theorem 1:

```
Theorem incidence_matrices_sc_product:
  forall (V:finType) (n:nat) (sc: {set (simplex V)}),
    simplicial_complex sc ->
      (incidence_mx_n sc n) *m (incidence_mx_n sc (n.+1)) = 0.
```

In the statement above, `*m` denotes the matricial product. The type information of each matrix includes its size. When the product operator is applied, the typechecking ensures that the two arguments have compatible sizes. Then the system knows the expected size of the result matrix and reads 0 as the null matrix of this size.

The formal proof of Theorem 1 follows the schema presented in Section 3. A large part of the proof is devoted to the work with summations, for which the Coq/SSReflect library "bigop" has played a key role.

For instance, the first summation splitting (equation (3)) is realized by:

```
rewrite (bigID (mem (boundary (enum_val j)))).
```

where j belongs to S_{n+1}.

The lemma `bigID` states that an iterated operation using a commutative monoidal operator can be split:

$$\sum_{i \in r | P_i} F_i = \sum_{i \in r | P_i \wedge a_i} F_i + \sum_{i \in r | P_i \wedge \sim a_i} F_i$$

It is also possible to split a summation (equation (6)) and at the same time rewrite the first resulting sum to 0 as in:

```
rewrite (bigID (mem (enum_val i))) big1.
```

`big1` states that, when a monoidal operator is iterated over elements that are all equal to the neutral, then the result is also the neutral element:

$$\sum_{i \in r | P_i} 0 = 0$$

Therefore, after the last tactic, the system will require a proof that all the terms of the first resulting summation are zero. `big1` is applied to obtain equations 4 and 7 of Section 3.

Our proof relies on two main reindexations: from ordinals to n-simplices (2) and later on from simplices to vertices (5). To perform the first reindexation, the script has the following shape:

```
rewrite (reindex_onto (enum_rank_in Hx0) enum_val) ; last first.
  by move=> x _ ; exact:enum_valK_in.
```

Where:

- Hx0 is a proof that there exists at least one n-simplex
- `enum_rank_in` enumerates the n-simplices since Hx0 ensures there is at least one
- `enum_val` enumerates the ordinals over which the sum is expressed
- `reindex_onto` reindexes from ordinals to n-simplices, given a bijection between both sets. Indeed, the second line proves that `enum_val` $\circ$ `enum_rank_in` = Id

The second reindexation is based on the injectivity of the face operator:

```
rewrite big_imset ; last exact:face_op_inj2.
```

Rewriting with the lemma `big_imset` triggers a check that the summation is expressed over the image of a set by a function. In our case, the system automatically infers that this function is the face operator `face_op`, and will then ask for a proof of its injectivity.

The lemma `eq_big` and its variants `eq_bigl` and `eq_bigr` allow to rewrite the predicate or the operand of an iterated operation. It is applied in particular to obtain equation 8 of Section 3:

```
rewrite (eq_bigr (fun _ => 1)).
```

The system will of course require a proof that the operand is equal to 1. Then it will rewrite the expression to a constant summation, allowing the use of the lemma `big_const` to replace it with a product (cardinal of the iterated set by the constant value).

Simple arithmetic arguments on cardinals will then complete the proof. The interested reader will find a snapshot of our development online [15].

6 Conclusions and Further Work

In this paper we have presented the formalization of simplicial complexes and their incidence matrices as well as the main theorem that gives meaning to the definition of homology groups. The proof assistant used has been Coq as well as the SSReflect extension and the libraries it provides. The verified algorithm is related to a Computer Algebra system for Algebraic Topology called Kenzo [9]. Therefore, our research is placed between the efforts to formalize mathematics and the application of formal methods in software systems.

Some parts of the future work are quite natural. The work presented here is solid enough to undertake the challenge of formalizing the construction of the Smith Normal Form [22] of incidence matrices, that is the diagonalization process which obtains homology groups of finite type objects.

Moreover, if we want to apply our Algebraic Topology methods to real life problems, for instance the study of medical images, we must be completely sure that our programs are safe. Therefore, the process to construct a simplicial complex from a digital image, presented in Section 3, should be formalized, too.

In addition, our proof seems generic enough to achieve the case of working with $\mathbb{Z}$-modules, instead of $\mathbb{Z}/2\mathbb{Z}$-modules, quite easily.

Another topic is related to the executability of our proofs, that is the computational capabilities of the objects we have defined (like the incidence matrices). Two main approaches are possible: code extraction or internal computations. The first one delivers a certified program and takes advantage of the existing extraction machinery of the Coq system. However, technical limitations have to be dealt with to get a usable program in our context. The second approach is somewhat more challenging regarding efficiency. Indeed, reaching inside Coq an execution speed on par with the one obtained by extraction and compilation is difficult because proofs cannot safely be erased from the terms (what extraction does). However, compilation techniques and evaluation strategies mitigating the performance impact are currently being studied and implemented.

One advantage of this second approach lies in the fact that it would enable the reuse of computational results in further formal developments. For instance, the computation of the smith normal form of a matrix could be used for further deductions, in the same system, on the topological object under study. We are currently studying the use of both code extraction and efficient computational techniques in the Coq/SSReflect system, applied to the objects and theories we have presented above.

References

1. Mathematical components team homepage, http://www.msr-inria.inria.fr/Projects/math-components
2. Aransay, J., Ballarin, C., Rubio, J.: A mechanized proof of the Basic Perturbation Lemma. Journal of Automated Reasoning 40(4), 271–292 (2008)
3. Ayala, R., Domínguez, E., Francés, A., Quintero, A.: Homotopy in digital spaces. Discrete Applied Mathematics 125, 3–24 (2003)
4. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development, Coq'Art: the Calculus of Inductive Constructions. Springer, Heidelberg (2004)
5. Bertot, Y., Gonthier, G., Biha, S., Pasca, I.: Canonical big operators. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 86–101. Springer, Heidelberg (2008)
6. Coq development team: The Coq Proof Assistant Reference Manual, version 8.3. Tech. rep. (2010)
7. Dey, T.K., Edelsbrunner, H., Guha, S.: Computational topology. In: Advances in Discrete and Computational Geometry. Contemporary Mathematics, pp. 143–190. AMS, Providence (1999)
8. Domínguez, C., Rubio, J.: Effective Homology of Bicomplexes, formalized in Coq. To appear in Theoretical Computer Science
9. Dousson, X., Sergeraert, F., Siret, Y.: The Kenzo program. Institut Fourier, Grenoble (1998), http://www-fourier.ujf-grenoble.fr/~sergerar/Kenzo
10. Gonthier, G.: Formal proof - The Four-Color Theorem. Notices of the American Mathematical Society, vol. 55 (2008)
11. Gonthier, G., Mahboubi, A.: A Small Scale Reflection Extension for the Coq system. Tech. rep., Microsoft Research INRIA (2009), http://hal.inria.fr/inria-00258384

12. Gonthier, G., Mahboubi, A., Rideau, L., Tassi, E., Théry, L.: A modular formalisation of finite group theory. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 86–101. Springer, Heidelberg (2007)
13. Gonzalez-Diaz, R., Medrano, B., Real, P., Sanchez-Pelaez, J.: Algebraic Topological Analysis of Time-Sequence of Digital Images. In: Ganzha, V.G., Mayr, E.W., Vorozhtsov, E.V. (eds.) CASC 2005. LNCS, vol. 3718, pp. 208–219. Springer, Heidelberg (2005)
14. Gonzalez-Diaz, R., Real, P.: On the Cohomology of 3D Digital Images. Discrete Applied Math. 147(2-3), 245–263 (2005)
15. Heras, J., Poza, M., Dénès, M., Rideau, L.: Incidence simplicial matrices formalized in SSReflect (2010), http://www.unirioja.es/cu/joheras/ismfissr/
16. Jacobson, N.: Basic Algebra II, 2nd edn. W. H. Freeman and Company, New York (1989)
17. Mackenzie, D.: Topologists and Roboticists Explore and Inchoate World. Science 8, 756 (2003)
18. MacLane, S.: Homology. Springer, Heidelberg (1963)
19. Orden, D., Santos, F.: Asymptotically efficient triangulations of the d-cube. Discrete and Computational Geometry 30(4), 509–528 (2003)
20. Rubio, J., Sergeraert, F.: Constructive Homological Algebra and Applications, Lecture Notes Summer School on Mathematics, Algorithms, and Proofs. University of Genova (2006), http://www-fourier.ujf-grenoble.fr/ sergerar/Papers/Genova-Lecture-Notes.pdf
21. Ségonne, F., Grimson, E., Fischl, B.: Topological Correction of Subcortical Segmentation. In: Ellis, R.E., Peters, T.M. (eds.) MICCAI 2003. LNCS, vol. 2879, pp. 695–702. Springer, Heidelberg (2003)
22. Veblen, O.: Analysis Situs. AMS Coll. Publ. (1931)
23. Wood, J.: Spinor groups and algebraic coding theory. Journal of Combinatorial Theory 50, 277–313 (1989)

Proof Assistant Decision Procedures for Formalizing Origami

Cezary Kaliszyk and Tetsuo Ida

Symbolic Computation Research Group,
University of Tsukuba
{kaliszyk,ida}@cs.tsukuba.ac.jp

Abstract. Origami constructions have interesting properties that are not covered by standard euclidean geometry. Such properties have been shown with the help of computer algebra systems. Proofs performed with computer algebra systems can be accompanied by proof documents, still they lack complete mathematical rigorousity, like the one provided by proof assistant checked proofs. Transforming such proofs to machine checkable proof scripts poses a number of challenges.

In this paper we describe issues that arise when proving properties of origami constructions using proof assistant decision procedures. We examine the strength of Gröbner Bases implementations comparing proof assistants with each other and with the implementations provided in computer algebra systems. We show ad-hoc decision procedures that can be used to optimize the proofs. We show how maximum equilateral triangle inscribed in a square construction can be formalized. We show how a equation system solving mechanism can be embedded in a CAS decision procedure of a proof assistant.

1 Introduction

1.1 Computational Origami

Origami is the traditional Japanese art of paper folding. In the recent years it is becoming more popular because of its applications in science, and education. It can also be a tool for geometrical constructions; namely it describes representing objects using paper folds. Instead of ruler and compass in Euclidean geometry one can use paper folding as a basis to create new points and lines on a surface. Starting with a square area represented by four corners of an initial origami, one can specify folds to superpose existing points or lines. Such superpositions give rise to new lines and the intersections of the new lines with existing ones create new points on the origami.

The traditional meaning of folds has been expressed in a rigorous way by Huzita and Justin [7,8,11]. Their work has classified all the possible folds, creating a system of operations for origami geometry (called *axioms* in the literature). They also show that the system is complete; this means that for the case of folding along one line no other operations that superpose points and lines are possible. The origami constructable points are a superset of the points constructable

J.H. Davenport et al. (Eds.): Calculemus/MKM 2011, LNAI 6824, pp. 45–57, 2011.

with Euclidean geometry; which also means that properties of the constructions performed with origami can be more complicated than those performed with ruler and compass.

The term *computational origami* has been used to refer to the branch of computer science that studies various aspects of origami. It includes representation of origami, design of origami algorithms, visualization, and geometrical theorem proving of computational and mathematical properties of origami. Computational origami most often refers to analyzing origami with a computer; while origami refers to an infinite virtual paper. In this paper the most important notion from computational origami used is the correspondence between geometry and algebra that allows to express the origami axiom system logically.

Recently a system for computational origami Eos has been created [15]. This system is capable of visualizing origami constructions based on Huzita's axioms, analysing the origami folds algebraically, and showing properties of the constructions. During the computational origami construction, geometrical constraints are accumulated. The symbolic representation is then transformed into algebraic forms by constraint solving using the generalized equations which describe the axioms. The algebraic representation is a set of polynomials, which describes the geometrical construction in a general way. The polynomials are then used to prove properties of the construction using the Gröbner bases method and CAD (Cylindrical Algebraic Decomposition). The proofs performed by Eos are automated; after specifying the goal, assumptions and a coordinate system, they use Mathematica's symbolic expression transformations and the implementations of the above two proof methods.

A number of constructions have been performed with Eos and certain properties of those constructions have been proved with its help:

- Trisection of an angle
- Maximum equilateral triangle
- Regular Heptagon
- Morley's Triangle
- Crane layers and sides

In this paper we use Eos as a source of computational origami problems; together with its mechanism for gathering the properties of origami constructions.

1.2 Computer Algebra functionality in Proof Assistants

Mainstream Computer Algebra Systems, for instance Mathematica and Maple, are weakly founded. This means that the expressions processed there do not have precisely defined semantics. The way expressions are processed in those systems is supposed to resemble mathematics as done on paper. But the fact that semantics are not well defined can be a reason for errors. There are various reasons for the mistakes found in mainstream CAS systems: assumptions can be lost, types of expressions can be forgotten [2], the system might get confused between branches of 'multi-valued' functions, and of course the algorithms of the

system themselves may contain implementation errors [21]. Simple mistakes have been found and fixed over the years; however mistakes made when performing more complicated computations are still found.

```
 In1 := vector [2; 2] - vector [1; 0] + vec 1
Out1 := vector [2; 3]
 In2 := diff (diff (λx. 3 * sin (2 * x) + 7 + exp (exp x)))
Out2 := λx. exp x pow 2 * exp (exp x) + exp x * exp (exp x) +
        -- 12 * sin (2 * x)
 In3 := N (exp (1)) 10
Out3 := #2.7182818284 + ...
 In4 := x + 1 - x / 1 + 7 * (y + x) pow 2
Out4 := 7 * x pow 2 + 14 * x * y + 7 * y pow 2 + 1
 In5 := sum (0,5) (λx. x * x)
Out5 := 30
 In6 := sqrt (x * x) assuming x > 1
Out6 := x
```

Fig. 1. Example session that shows interaction with the prototype computer algebra input-response loop. When the user inputs expressions to be processed in the `In` fields, the system produces output in `Out` lines together with (not displayed) HOL Light theorems that state the equality between the inputs and the outputs. Because of the way numbers are defined in HOL Light, working with them requires coercions. To work with multiple types in HOL Light coercions are needed and the `&` symbol is necessary for some numbers; here prioritization has been assumed and the coercions were skipped. Similarly the rest of a numerical approximation (marked as ...) is hidden in the output.

We have built a prototype computer algebra like input-response-loop inside HOL Light [13], with the user interface designed close to the interfaces of popular computer algebra systems. In Figure 1 we show examples of simplifications that it can perform automatically, for example:

- Real analysis and transcendental functions, including symbolic differentiation and integration
- Simple complex computations
- Approximations, decimal approximations and rounding
- Binomials, permutations, primality
- Vectors, matrix operations

Implementing the computer algebra inside a proof assistant with all the simplifications certified by the latter guarantees that the system will make no mistakes [13].

This architecture also has drawbacks, the two main drawbacks are the complexity of the implementation and the efficiency. Every simplification that one needs to perform has to be a proof producing simplification and every equation needs to be accompanied with a proof. Similarly all the simplifications as seen in Figure 1 are relatively simple; performing any more complicated operations would become too slow in the current prototype. Also, when talking about the efficiency of a computer algebra system has to take into account scalability.

The formulas that show up when proving properties of origami systems can be processed in a commercial computer algebra system; however already there it can take a substantial amount of time. The Eos proof document describing the proof for Morley's theorem by Abe's method [5] shows that the call to Mathematica's Gröbner bases algorithm takes 1668.12 seconds on a modern machine. Taking into account that all operations of the prototype CAS system are much slower than their Mathematica counterparts, and the fact that the implementations of algorithms chosen to be certified are less efficient algorithms it is naturally not possible to obtain the same proof performed automatically in the prototype system.

In this paper we show tactics that allow simplifying the goals that arise in Eos in such a way that they can be fully formally verified with the help of decision procedures. The tactics solve (or simplify) systems of equations. We compare the efficiency of the implementations of computer algebra algorithms in various proof assistants and show how the goals that arise in certain origami constructions can be proved with the help of these tools.

1.3 Contents

The rest of this paper is organized as follows. In Section 2 we describe proof documents generated by Eos. In Section 3 we describe how the algebraic proof of the equilateral triangle construction can be performed manually in a proof assistant. In Section 4 we show a tactic that preprocesses the goals for the proof assistant decision procedures. Section 5 presents related work. Finally in Section 6 we give a conclusion and present possible future work.

2 Formalizing a Proof Document

When an origami construction is performed with the help of Eos and its properties are checked, optionally the system can create a *proof document*. It is an output that includes all the operations performed within an origami construction along with all the steps of the proof. Every proof document starts with a header and a copy of the user given program for constructing the origami. The program always starts with a new origami and includes the performed folds and unfolds and the new points and lines. For every new point and line the way it is obtained is stored.

In the first stage of the proof document the points and lines are represented in a geometrical way. This means that geometric relations between points and lines are given, but no coordinates are assigned to the points.

In the second stage the points and lines are given variable coordinates and with the help of the origami axiom system the geometrical properties (properties that talk about points, lines and folds) are translated into algebraic properties. The operations of origami constructions are expressed in terms of equations (assumed as axioms) and this step is a substitution in these equations. During this stage some precision is lost. The equations that talk about equal distances would be expressed as equality between square roots. To avoid square roots that are not handled by real arithmetic decision procedures, those equations are squared.

There are two kinds of optimizations performed in the second stage. First, same points and lines are assigned the same coordinates. Second, predicates that are not relevant for the goal are removed. These two steps have been introduced since Mathematica's algorithm is not able to eliminate them automatically and otherwise its complexity increases with a bigger number of variables and equations. In a formalization irrelevant assumptions and equal variables can be easily removed (for example in Isabelle the `clarify` tactic removes both).

The last part of the proof document is the crucial one from our point of view. The decision procedures built into Mathematica are used to verify the large formula prepared in the previous steps. The computation performed in this step is done behind the scenes and as such cannot be verified independently. This is the reason for a formalization of this step, which we describe in the following section.

3 Equilateral Triangle Construction

In this section we will focus on the proof of the maximal equilateral triangle construction performed with origami, and the formalization of the computational part of the equilateral proof. The construction as shown in Figure 2 is described in more detail in paper [14]. The computational origami system Eos gathers the geometrical properties of the origami lines and points and translates them to algebraic properties. Similarly the goal (the fact that the triangle is equilateral) is expressed in terms of an algebraic constraint. The formula to be proved at the end of this step is given in Figure 3.

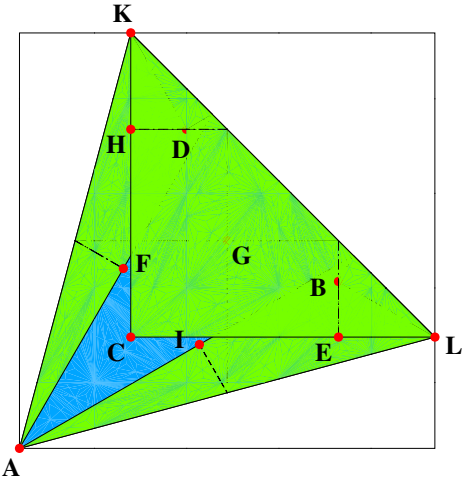


Fig. 2. Equilateral triangle construction as performed by Eos. Only the final origami state is shown. For the folds where more than one resulting line is possible, the system interactively asks the user which of the lines is the intended one.

$$a_1 = 0 \wedge (-1+a_1)*(-1+b_1) = 0 \wedge (-1+b_1)*b_1 = 0 \wedge (-1+a_2)*(-1+b_2) = 0 \wedge (-1+b_2)*b_2 = 0 \wedge (-1+a_3)*(-1+b_3) = 0 \wedge (-1+b_3)*b_3 = 0 \wedge (-1+a_4)*(-1+b_4) = 0 \wedge (-1+b_4)*b_4 = 0 \wedge (-1+b_4)*(b_1+c_1+2a_1) = 0 \wedge (b_1+2*c_1)/2 = 0 \wedge c_1 - b_1 + 2*a_1 + a_4^2*b_1 + a_4^2*c_1 + (-2)*a_1*b_4 + 2*a_1*a_4^2 + 2*a_4*b_1 + 2*b_1*b_4 = 0 \wedge (-1+b_3)*(a_2+c_2) = 0 \wedge c_2 - a_2 + a_2*a_3^2 + a_3^2*c_2 + 2*a_2*a_3 + 2*a_2*b_3 + 2*b_2*b_3 = 0 \wedge a_3 + b_3 + c_3 = 0 \wedge a_4 + b_4 + c_4 = 0 \wedge x_3 = 0 \wedge x_8 = 0 \wedge b_1*x_1 - a_1*y_1 = 0 \wedge c_1 + (a_1*x_1)/2 + (b_1*y_1)/2 = 0 \wedge b_1*(-1+x_2) - a_1*y_2 = 0 \wedge a_2*y_2 + b_2*x_1 - a_2*y_1 - b_2*x_2 = 0 \wedge c_1 + a_1*(1+x_2)/2 + (b_1*y_2)/2 = 0 \wedge (2*c_2 + a_2*x_1 + a_2*x_2 + b_2*y_1 + b_2*y_2)/2 = 0 \wedge c_1 + a_1*x_3 + b_1*y_3 = 0 \wedge c_1 + a_1*x_4 + b_1*y_4 = 0 \wedge c_2 + a_2*x_4 + b_2*y_4 = 0 \wedge -1 + y_5 = 0 \wedge c_2 + a_2*x_5 + b_2*y_5 = 0 \wedge a_1*(y_5 - y_6) + b_1*(x_6 - x_5) = 0 \wedge c_1 + a_1*(x_5+x_6)/2 + b_1(y_5+y_6)/2 = 0 \wedge y_7 = 0 \wedge c_3 + a_3*x_7 + b_3*y_7 = 0 \wedge c_4 + a_4*x_8 + b_4*y_8 = 0 \wedge -1 + (x_8 - y_8 + x_7(-1+y_8) + (1-x_8)*y_7)*\xi_1 = 0 \Rightarrow -((1-x_7)^2) + (x_7-x_8)^2 - (1-y_7)^2 + (y_7-y_8)^2 = 0 \wedge (x_7-x_8)^2 + (y_7-y_8)^2 - (1-x_8)^2 - (1-y_8)^2 = 0$$

Fig. 3. Algebraic formula to be proved. The equations arising from all the steps of the construction have been gathered by Eos. We show it here, as an example problem to be solved; the equation systems for other origami problems are similar, just contain of more equations with more variables.

The computer-algebra decision procedure as presented in [13] is not able to do any simplifications on a formula this big. Still parts of it can be simplifed and this can be made the first part of the proof. We performed this first part in three proof assistants to compare the available mechanisms for reasoning on goals arizing from Eos; we also compare it with Mathematica's simplification. For the experiment we used HOL Light [6], Isabelle [19] and Coq [3].

The Gröbner bases algorithms present in the 3 proof assistants are too weak to solve the goal without any preprocessing. In case of HOL Light and Isabelle the mechanisms do not terminate in reasonable time (we tried one week on a powerful server), in case of Coq after a few minutes the procedure exits with the goal transformed in ring equations but not solved.

To perform the complete proof in HOL Light we first need to simplify the equations. To do this manually we first solve some of the sub-equations using the ring and field decision procedures. The `COMPLEX_FIELD` conversion will produce the rewrite rules for solving linear equations and `FIX_X_ASSUM` can be used to replace the original equation assumption with the solved one. The HOL Light goal-state is internally a sequence of implications, where the goal is implied by all the assumptions. To rewrite all variable occurrences in the other assumptions, a specific solved assumption is the only one kept in the assumptions list while all other ones are moved to the goal.

Solving linear equations and performing substitutions allows reducing the goal to a system of 5 equations. Moving the assumptions back to the goal and generalizing over the five variables creates two subgoals, which both can be solved by HOL Light's built in real quantifier elimination procedure. The procedure takes 18 seconds to prove the two goals on a modern computer. The HOL Light proof script is 98 lines long.

The same proof can be performed in a similar way in Isabelle; with two important differences. The `clarify` tactic together with the simplifier is able to substitute the value of a computed assumption in other ones and remove unnecessary assumptions automatically. Similarly the `algebra` decision procedure

is able to operate on a goal with variables and assumptions introduced in the context as well as with a goal being a conjunction of two equations. This means that the manual proof can be cleaner than the HOL Light one; still the choice of equations to solve and the rules to apply is left to the user. The final call to the Gröbner bases decision procedure takes 1.4 seconds to solve the simplified goal. The Isabelle proof script is 151 lines long.

In our experiments the `nsatz` tactic in Coq was too weak to solve even the goal with five equations. This result was quite surprising, as we expected the procedure that makes use of reflection to be stronger than the ones present in HOL Light or Isabelle. The procedure is able to solve goals involving quadratic equations, but for more complicated goals it transforms the goal into equations mentioning explicitly the real ring and exits.

4 Equations Solving Tactic

In this section we describe a tactic that solves or simplifies systems of equations motivated by the manual formalization. With the help of this tactic the original set of equations can be simplified to a much simpler formula, namely for the problem presented in the previous section it can be simplified automatically to the one presented in Figure 4.

$$a_4^2 + 4 * a_4 + 1 = 0 \wedge a_3^2 + 4 * a_3 + 1 = 0 \wedge 1 + a_3 - a_3 * x_7 = 0 \wedge 1 - y_8 + a_4 = 0 \wedge -1 - xi_1 * y_8 - \xi_1 * x_7 + \xi_1 * x_7 * y_8 = 0 \Rightarrow -2 + y_8^2 + x_7 * 2 = 0 \wedge -2 + y_8 * 2 + x_7^2 = 0$$

Fig. 4. Simplified, with solved variables removed for clarity

The tactic is similar to extended Gaussian elimination; it allows the original system to contain equations of higher degrees as well as inequalities. The simplifications are performed in an order that takes into account the complexity of symbolic transformations. Given a system of equations the tactic counts the numbers of occurrences of variables in each equation. As the tactic is running a threshold for a maximum number of variables is increasing. For the equations that have less variables than the current threshold, all the occurrences are moved to one side of the equation. Next the left side of the equation can be normalized, and in case if the equation is a linear equation it can be solved.

In the following we will distinguish the assumptions and goal part of the HOL Light goal-state, as described in Section 3. The tactic analyses the goal and performs the operations according to Fig. 5:

The tactic optionally is able to remove the variables that have been solved. Since the equations are processed together with the goal, the variables that have been solved have also been replaced in the goal. This means that removing equations involving those variables does not weaken the goal.

Repeating the tactic performs a simplification of the original system of equations. For a system of linear equations the procedure is equivalent to Gaussian elimination optimized to perform fewer operations. It also accepts higher order equations and inequalities, performing normalization of higher degree equations.

1. Start with threshold t equal 1.
2. If t is greater than maximum threshold exit.
3. Find assumptions with number of free variables less than t. Move those assumptions to the assumptions part of the goal-state using `DISCH_TAC` and keep the other assumptions in the goal part using `UNDISCH_TAC`.
4. If there are no assumptions, increase t by 1 and go to step 2.
5. If there are no variables in the assumptions, simplify all the assumptions with `COMPLEX_POLY_CONV`, increase t and go to step 2.
6. For each equation select a variable to normalize
   (a) Choose a variable that has only one linear factor equal 1
   (b) If not possible choose a variable with only linear factors
   (c) If not possible choose a variable with the lowest degree
7. In every equation bring all occurrences of the variable to normalize to one side and the other factors on the other and call `COMPLEX_POLY_CONV` to simplify both sides.
8. If there are equations where only a linear factor remains, divide both sides by the linear factor and simplify the goal and the other equations with it. Set t equal 1 and go to step 3.
9. Increase t by 1 and go to step 2.

Fig. 5. The simplification algorithm performed by the tactic

4.1 Evaluation

We tested the tactic on four proofs. The equilateral triangle construction gives rise to two obligations: the proof that the triangle is indeed equilateral and the proof that it is a maximal triangle inscribed in the original square of the origami. We also looked Eos regular heptagon construction and the Morley's triangle construction proofs.

We believe these are the most relevant and interesting proofs from origami theorem proving that can be analyzed with algebra only. There are two other important origami proofs performed with Eos that we cannot deal with our automatic approach, namely the proof about the sides and layers of the crane origami construction and the proofs about angle trisection constructions. The reason for this is that the first kind includes three dimensional properties that cannot be easily encoded with the same kind of algebraic formulas and the second requires trigonometry.

In all the proofs that we have tried, the tactic did reduce the number of equations and inequalities and the number of quantified variables. In case of the equilateral proof it was able to reduce the 35 equations to 5 equations. For the maximal proof it reduced 37 (in)equations to 9 (in)equations. For the regular heptagon proof it reduced 63 equations to 12 equations. In case of the Morley's triangle construction the tactic would only eliminate 8 simple linear equations but would not be able to progress further because of a bit amount of inequalities. This construction has 27 cases and only for 9 of them the triangle is indeed a Morley's triangle. This means that there are 6 inequalities that limit the effectiveness of the tactic.

For all the above examples the tactic run-time is less than one second on a modern computer. Only in the equilateral triangle proofs the Gröbner bases decision procedure is able to finish the proof automatically. This takes 18 seconds.

In case of the regular heptagon construction, preprocessing reduces an original system of 63 equations to 12 equations. At this stage the Gröbner decision procedures present in the proof assistants are still too weak to finish the goal and additional manual solving of higher-level equations is needed. Finishing the proof manually is complicated, since processing higher order equations manually is cumbersome. When processing the equations in a computer algebra system, irrational numbers and their linear factors are automatically moved from denominators to numerators by multiplying the equations by appropriate factors.

4.2 Inequalities

During the construction of an origami, some of the fold operations have a number of solutions. When the user interactively requests such a fold in an Eos construction, the system will in turn ask which of the two of three fold lines is the intended one. This is then turned in a geometric constraint that a point is included in a certain segment.

Similarly inequalities may come from assumptions about points being on the origami (some approaches to computational origami assume that the original paper is infinite; however after performing folds the area on which points can be considered becomes bounded) or even from the property to be proved (for example the case of a constructed triangle being the maximal one). Such constraints are naturally easier provided as inequalities and therefore cannot be directly used in conjunction with the Gröbner bases algorithm.

McLaughlin and Harrison [16] have implemented a proof producing CAD algorithm that follows the Hörmander's CAD algorithm. We tested the efficiency of the algorithm for practical problems and unfortunately it is too slow to be used; in fact we were not able to solve problems of size 3. However the procedure comes useful when solving second order equations. Still to be useful in a manual formalization, additional tactics for normalizing equations with irrational fractions would be needed.

5 Related Work

As a part of the ForMath project, a proof producing CAD in Coq has been implemented [4]. Its efficiency is however not enough to perform proofs as shown in this paper. We also tried the `nsatz` tactic present in Coq [20]. It has the advantage of using reflection to obtain very efficient proofs. Still many of the origami goals require complex numbers and the tactic does not work with those; also for the proofs that can be performed with real numbers (for example the simplified system of 5 equations for the equilateral triangle proof) the tactic is not able to solve the goal.

A Common-Lisp implementation of the Buchberger algorithm has been verified in ACL2 [17]. There are numerous bridges between proof assistants and

computer algebra systems (for example [1]). Such architectures have different degree of trust; the proof of the computer algebra is either used as an oracle or as a hint and recomputed. In this research we aim at a complete formalization done in a proof assistant.

There are many programs for proving geometrical properties. Certain programs are able to export goals to proof assistants to be able to formally certify the constructions. Examples of those are GeoProof [18] or GCL [10]. We have not known of any origami proofs performed in this way.

6 Outlook

There are many decision procedures for real and complex arithmetic present in proof assistant libraries and extensions. They are certainly useful, but they still require special goal shapes and are very slow in comparison with the algorithms present in computer algebra.

We have shown how the goals that arise in origami proofs can be automatically simplified and some of those can be immediately solved by proof assistant decision procedures possibly with minor manual interaction. We compared the efficiency of implementations of Gröbner bases algorithm in HOL Light, Isabelle and Mathematica.

The developed equation system simplifying tactic is available as a part of the CAS-conversion [12]. As such it is both available in the prototype certified computer algebra system inside HOL Light, and as a part of the general CAS simplification procedure.

6.1 Future Work

In this paper we assumed the correctness of the part of the Eos system that gathers the geometric conditions and translates them to an algebraic formula. Origami operations are usually expressed in terms of numbers of possible fold lines that satisfy certain constraints. For a proof assistant development, for every operation equivalent algebraic constraints could be proved. The original constraints of the operations are expressed using geometry, and it is also possible to transform those into geometrical constraints.

The principles of folding could be formalized in a proof assistant together with the basic properties of the fold lines that are implied by particular folds. Those properties have been proved geometrically only on paper (for example in [9]). Creating a development like this would be a basis for a formalized origami theory. For operations that give rise to a single possible fold line, this is straightforward; however certain origami fold operations are known to be equivalent to solving higher order equations. In the Equilateral triangle construction we use two of the Huzita's fold principles: `fold2` and `fold5`. The first of those superposes one point onto a different point creating only one fold line. However the latter one superposes one point onto a line and making the crease pass through another point (tangent from a point to a parabola) giving two possible fold lines in the

general case. When the user performs such a fold interactively, Eos presents the user with a choice visually. In a proof assistant, one needs a formal statement that will be known to limit the equation to a single solution.

In case of the equilateral triangle construction we can limit the result of a fold operation by adding intersection constraints. With such a mechanism we can imagine the formal statement in a proof assistant to look like on Figure 6. Here both of the `fold5` operations give rise to 2 fold lines, and an additional intersection constraint is necessary to make the construction unique and to make the goalstate true. In case of more complicated constructions intersection may not always be enough to specify a unique fold line and a more elaborate mechanism of specifying a unique fold line will be necessary.

lemma equilateral_triangle:
fixes
A B C D E F G H J K :: point
and k l m n :: line
assumes
a1: "A = (0, 0)"
and a2: "B = (1, 0)"
and a3: "C = (1, 1)"
and a4: "D = (0, 1)"
and a5: "k = fold2 A D"
and a6: "E = intersect k A D"
and a7: "F = intersect k B C"
and a8: "l = fold2 A B"
and a9: "H = intersect l D C"
and aa: "G = intersect l E F"
and ab: "m ∈ fold5 D l A"
and ac: "J = intersect m C D"
and ad: "n ∈ fold5 B k A"
and ae: "K = intersect n B C"
shows
"dist A J = dist J K"
and "dist J K = dist A K"

Fig. 6. With a formalization of the origami axiom system, a sequence of unique folds and intersections can be described in a goalstate

References

1. Adams, A., Dunstan, M., Gottliebsen, H., Kelsey, T., Martin, U., Owre, S.: Computer algebra meets automated theorem proving: Integrating Maple and PVS. In: Boulton, R.J., Jackson, P.B. (eds.) TPHOLs 2001. LNCS, vol. 2152, pp. 27–42. Springer, Heidelberg (2001)

2. Aslaksen, H.: Multiple-valued complex functions and computer algebra. SIGSAM Bulletin (ACM Special Interest Group on Symbolic and Algebraic Manipulation) 30(2), 12–20 (1996)
3. Barras, B., Boutin, S., Cornes, C., Courant, J., Coscoy, Y., Delahaye, D., de Rauglaudre, D., Filliâtre, J.C., Giménez, E., Herbelin, H., et al.: The Coq Proof Assistant Reference Manual Version 8.3. LogiCal project (2010), http://coq.inria.fr/refman/
4. Cohen, C., Mahboubi, A.: A formal quantifier elimination for algebraically closed fields. In: Autexier, S., Calmet, J., Delahaye, D., Ion, P.D.F., Rideau, L., Rioboo, R., Sexton, A.P. (eds.) AISC 2010. LNCS, vol. 6167, pp. 189–203. Springer, Heidelberg (2010)
5. Ghourabi, F., Ida, T., Kasem, A.: Proof documents for automated origami theorem proving. Submitted to ADG 2011 post-proceedings (2011)
6. Harrison, J.: HOL Light: A tutorial introduction. In: Srivas, M., Camilleri, A. (eds.) FMCAD 1996. LNCS, vol. 1166, pp. 265–269. Springer, Heidelberg (1996)
7. Huzita, H.: Axiomatic development of origami geometry. In: Huzita, H. (ed.) Proceedings of the First International Meeting of Origami Science and Technology, pp. 143–158 (1989)
8. Huzita, H.: The trisection of a given angle solved by the geometry of origami. In: Huzita, H. (ed.) Proceedings of the First International Meeting of Origami Science and Technology, pp. 195–214 (1989)
9. Ida, T., Ghourabi, F., Kasem, A., Kaliszyk, C.: Towards theory of origami fold. Submitted to ISSAC (2011)
10. Janicic, P.: Geometry constructions language. J. Autom. Reasoning 44(1-2), 3–24 (2010)
11. Justin, J.: Résolution par le pliage de l'équation du troisième degré et applications géométriques. In: Huzita, H. (ed.) Proceedings of the First International Meeting of Origami Science and Technology, pp. 251–261 (1989)
12. Kaliszyk, C.: Prototype computer algebra system in HOL Light, http://score.cs.tsukuba.ac.jp/~kaliszyk/holcas.php
13. Kaliszyk, C., Wiedijk, F.: Certified computer algebra on top of an interactive theorem prover. In: Kauers, M., Kerber, M., Miner, R., Windsteiger, W. (eds.) MKM/CALCULEMUS 2007. LNCS (LNAI), vol. 4573, pp. 94–105. Springer, Heidelberg (2007)
14. Kasem, A., Ida, T.: Computational origami environment on the Web. Frontiers of Computer Science in China 2(1), 39–54 (2008)
15. Kasem, A., Ida, T., Takahashi, H., Marin, M., Ghourabi, F.: E-origami system Eos. In: Proceedings of the Annual Symposium of Japan Society for Software Science and Technology, JSSST, Tokyo, Japan (September 2006)
16. McLaughlin, S., Harrison, J.: A proof-producing decision procedure for real arithmetic. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 295–314. Springer, Heidelberg (2005)
17. Medina-Bulo, I., Palomo-Lozano, F., Ruiz-Reina, J.-L.: A verified Common Lisp implementation of Buchberger's algorithm in ACL2. J. Symb. Comput. 45(1), 96–123 (2010)

18. Narboux, J.: A graphical user interface for formal proofs in geometry. J. Autom. Reasoning 39(2), 161–180 (2007)
19. Nipkow, T., Paulson, L.C., Wenzel, M.T. (eds.): Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
20. Pottier, L.: Connecting Gröbner bases programs with Coq to do proofs in algebra, geometry and arithmetics. In: Rudnicki, P., Sutcliffe, G., Konev, B., Schmidt, R.A., Schulz, S. (eds.) LPAR Workshops. CEUR Workshop Proceedings, vol. 418, CEUR-WS.org (2008)
21. Critique of the Mathematical Abilities of CA Systems. In: Wester, M. (ed.) Contents of Computer Algebra Systems: A Practical Guide, John Wiley & Sons, Chichester (1999)

Using *Theorema* in the Formalization of Theoretical Economics*

Manfred Kerber[1], Colin Rowat[2], and Wolfgang Windsteiger[3]

[1] Computer Science
[2] Economics
University of Birmingham, Birmingham B15 2TT, England
[3] RISC
JKU Linz, 4232 Hagenberg, Austria

Abstract. Theoretical economics makes use of strict mathematical methods. For instance, games as introduced by von Neumann and Morgenstern allow for formal mathematical proofs for certain axiomatized economical situations. Such proofs can—at least in principle—also be carried through in formal systems such as *Theorema*. In this paper we describe experiments carried through using the *Theorema* system to prove theorems about a particular form of games called pillage games. Each pillage game formalizes a particular understanding of power. Analysis then attempts to derive the properties of solution sets (in particular, the core and stable set), asking about existence, uniqueness and characterization.

Concretely we use *Theorema* to show properties previously proved on paper by two of the co-authors for pillage games with three agents. Of particular interest is some pseudo-code which summarizes the results previously shown. Since the computation involves infinite sets the pseudo-code is in several ways non-computational. However, in the presence of appropriate lemmas, the pseudo-code has sufficient computational content that *Theorema* can compute stable sets (which are always finite). We have concretely demonstrated this for three different important power functions.

1 Introduction

Theoretical economics may be regarded as a branch of applied mathematics, drawing on a wide range of mathematics to explore and prove properties of stylized economic environments. Since the Second World War, one particularly important body of theory has been game theory, as introduced by von Neumann and Morgenstern [17] and successfully developed by John Nash.

* The first author would like to thank the 'Bridging The Gap' initiative under EPSRC grant EP/F033087/1 for supporting this work. The first and second author would like to thank the JKU Linz for its hospitality.

J.H. Davenport et al. (Eds.): Calculemus/MKM 2011, LNAI 6824, pp. 58–73, 2011.

The game theory stemming from von Neumann and Morgenstern has become known as cooperative game theory; it allows abstraction from the details of how agents might interact, instead focusing directly on how final outcomes may or may not dominate each other. As dominance is a binary relation, cooperative game theory has lent itself naturally to axiomatic analyses. The game theory stemming from Nash has become known as non-cooperative game theory; it is explicitly constructive, requiring specification of a game form that details the set of permissible moves available to agents. Solutions to cooperative games have been difficult to calculate relative to non-cooperative games, contributing to the latter body of theory's current preeminence within game theory.[1]

In the current work, we use *Theorema* [19] to formalize a particular cooperative game form, called pillage games, and to prove formally certain properties of them. Pillage games, introduced in [8], form an uncountable set of cooperative games, taking the two best known classes of cooperative games (those in characteristic and partition function form) as boundary points. At the same time, even though each is defined over an uncountable domain, their structure has thus far been sufficient to avoid Deng and Papadimitriou's [5] pessimistic conclusions that whether a solution even exists may be undecidable. Thus, pillage games provide a class of games for analysis that is simultaneously rich and tractable. To our knowledge, this represents the first attempt to formally prove properties of a cooperative game. Related is work on other axiomatic proofs within economic theory which have been formalized. Arrow's theorem in social choice has attracted the most attention, including studies by Wiedijk [18] using *Mizar*, Nipkow [15] using *HOL*, and Grandi and Endriss [6] using *Prover9*. Non-cooperative game theory has also received attention, including by Vestergaard and co-authors [16] with *Coq*.

The formalization of a particular game form in economics can be interesting to computer science for at least two reasons, and to economics for at least one. For computer science, economics is a relatively new area for automated theorem proving and therefore presents a new set of canonical examples and problems. Secondly, it is an area which typically involves new mathematics in the sense that axioms particular to economics are postulated. Further, in the case of pillage games, the concepts involved are of a level that an undergraduate mathematics student can understand easily. That is, the mathematics is of a level that should be much more amenable for formalizations than research level mathematics.

For economics, as in any other mathematical discipline [11], establishing new results is typically an error-prone process, even for the most respected researchers. We cite but two examples from cooperative game theory:

1. In founding game theory, von Neumann and Morgenstern [17] assumed that one of the key concepts of the field, the so-called stable set (by them just called the "solution"), always existed in games in characteristic function form. This was subsequently demonstrated by Lucas [12] to be incorrect.

[1] For example, Gambit [14] solves a broad class of non-cooperative games.

2. Nobel Prize winning economist and game theorist Maskin [13] claimed that certain properties of a game in partition function form extended from $n = 3$ to $n > 3$. However, de Clippel and Serrano [4] found counterexamples with $n > 3$.

It is understandable that such problems occur since typically for any new axiom set humans have initially no or only limited intuition. This way it is easy to assume false theorems and to overlook cases in proofs. Proofs found in mathematics in general and theoretical economics in particular, can be viewed from a logical point of view more like proof plans. That is, not all details are given, hidden assumptions may be overlooked, proof steps may be incorrect, generalizations may not hold. Thus, any mathematical discipline, including theoretical economics, can benefit from formalizing proofs since this will make proofs much more reliable. However, there are other potential benefits. For instance, in experimenting with axiomatizations it is much easier to reuse proof efforts. Furthermore the dependencies of assertions can be accessed more easily and experiments with the computational content of theorems becomes possible which without computer support would be time consuming and error-prone.

In this work we report on our experiments with the *Theorema* system which we conducted to formalize pillage games, to prove certain properties of them, and to exploit computational features in them. Full verification of the formal statements is an important goal of these experiments. However, we also look at less labour extensive ways of making use of *Theorema* and will discuss this.

The paper has the following structure. In the next section we give a brief introduction to pillage games. In Section 3 we present the representations in *Theorema* and the proofs of some formal statements which we formally proved in *Theorema*. A focus of the presentation as presented in Subsection 3.4 is the pseudo-code, which summarizes the results of the paper we formalize. This pseudo-code is non-computational in several ways. However, a mixture of proof and computation makes it possible to evaluate it in concrete cases. In Section 4 we evaluate the approach taken.

2 A Brief Introduction to Pillage Games

The class of games used for our experiments are called pillage games, a particular form of cooperative games, introduced by Jordan in [8]. In a nutshell, pillage games describe how *agents* (n in total) can form coalitions in order to redistribute all or part of the possessions of other coalitions among themselves. The possessions are described by a so-called *allocation*, a vector of n non-negative numbers which sum to one. Given two such vectors, x and y, three coalitions are induced: the *win set* of a transition from x to y is $\{i \mid y_i > x_i\}$; the *lose set* is $\{i \mid x_i > y_i\}$; the remaining agents are indifferent between x and y. Pillage is possible if and only if the win set is more powerful at x than is the lose set; if this is so, it is said that y *dominates* x.

The power of a coalition is determined by a so-called *power function*, π, a function which depends exclusively on the coalition members and the holdings of all agents. A power function must satisfy three monotonicity axioms: firstly *weak coalition monotonicity (WC)*, whereby taking a new member into a coalition does not decrease the coalition's power; secondly *weak resource monotonicity (WR)*, whereby weakly increasing the holdings of a coalition's members does not decrease the coalition's power; and thirdly, *strong resource monotonicity (SR)*, whereby strictly increasing the holdings of a coalition's members strictly increases the coalition's power. This setting given, two sets are of interest. Firstly, the *core*, the set of undominated allocations, and secondly the *stable set*, a set of allocations such that none dominates another (called *internal stability*) but at least one dominates each non-member allocation (called *external stability*). The core always exists, and is unique, but may be empty. A stable set may not exist since it has to satisfy two conflicting properties, on the one hand it must contain sufficiently many elements so that any element not in it is dominated by an element in it (e.g. the empty set is not externally stable), on the other hand it must not contain so many elements that none dominates another (e.g. the full set of all allocations is not internally stable in a pillage game). If a stable set exists, it is finite and contains the core; uniqueness has been established for some pillage games, and no counterexamples have been found as yet. If agents are forward looking, expecting that y dominating x may allow a subsequent comparison between z and y. Jordan [8] proved that a stable set is a *core in expectation*, a set of undominated allocations given some such future expectation (and consequent comparison of x and z, rather than x and y).

Jordan [8] proved general properties about pillage games (such as the finiteness of the stable set) and investigated the possibilities of three particular power functions for arbitrarily many agents. Kerber and Rowat [10] studied an infinite class of power functions for three agents. In particular they gave a complete characterization of the stable set for arbitrary power functions (with three agents) which satisfy three additional axioms. The first axiom is *continuity* in the resources; although definable with an ϵ-δ-statement, the intermediate value theorem is actually used. The second axiom, *responsiveness*, requires that a coalition gaining a member with some power as a singleton strictly increases the coalition's power. The third axiom, *anonymity*, requires that power, dominance, and—consequently—the stable set are invariant under permutations of the agents; thus, an agent's identity is irrelevant to a coalition's power, merely its presence or absence, and its holdings are relevant.

3 Formalizations in *Theorema*

Within this case study, we fully proved the first three lemmas from [10]. Furthermore, we give a formalization of the pseudo-code that summarizes the results derived in that paper. In this section, we briefly give the main assertions,

which we proved in *Theorema* input syntax.[2] We begin with the definition of a power function. In all what follows, $I[n] := \{1, \ldots, n\}$ stands for the set of n agents and $X[n]$ denotes the set of all allocations for n agents.

With these preliminaries we use *Theorema* to formally define *weak coalition monotonicity (WC)*, *weak resource monotonicity (WR)*, *strong resource monotonicity (SR)*, and *power function* as follows.[3]

Definition. ["WC", any$[\pi, n]$, bound[allocation$_n[x]$],

$$\mathrm{WC}[\pi, n] :\Leftrightarrow n \in \mathbb{N} \wedge \left[\forall_{\substack{C1, C2 \\ C1 \subset C2 \wedge C2 \subseteq I[n]}} \; \forall_{x} \; \pi[C2, x] \geq \pi[C1, x] \right]$$

Definition. ["WR", any$[\pi, n]$, bound[allocation$_n[x]$, allocation$_n[y]$],

$$\mathrm{WR}[\pi, n] :\Leftrightarrow n \in \mathbb{N} \wedge \left[\forall_{\substack{C \\ C \subseteq I[n]}} \; \forall_{x,y} \left(\left(\forall_{i \in C} \; y_i \geq x_i \right) \implies \pi[C, y] \geq \pi[C, x] \right) \right]$$

Definition. ["SR", any$[\pi, n]$, bound[allocation$_n[x]$, allocation$_n[y]$],

$$\mathrm{SR}[\pi, n] :\Leftrightarrow n \in \mathbb{N} \wedge \left[\forall_{\substack{C \\ C \subseteq I[n] \wedge C \neq \emptyset}} \; \forall_{x,y} \left(\left(\forall_{i \in C} \; y_i > x_i \right) \implies \pi[C, y] > \pi[C, x] \right) \right]$$

Definition. ["powerfunction", any$[\pi, n]$, powerfunction$[\pi, n] :\Leftrightarrow \wedge \left\{ \begin{array}{l} \mathrm{WC}[\pi, n] \\ \mathrm{WR}[\pi, n] \\ \mathrm{SR}[\pi, n] \end{array} \right.$]

In this formalization, we decided not to put explicit conditions on variables in definitions assuming that defined expressions will only be used "as intended

[2] The *Theorema* language syntax comes very close to how mathematicians are used to write up things. In particular, two-dimensional notation can be used both in input and output through *Mathematica*'s notebook technology, so that a *Theorema* formalization can easily be read and understood by a mathematician. In this presentation, we typeset all *Theorema* expressions in LaTeX with the aim to mimick their appearance in *Theorema* as closely as possible. Formal text blocks (definitions, theorems, lemmas, etc.) in *Theorema*, so-called environments, are of the form '*Env*[*l*, any[*v*], with[*C*], bound[*r*], *form*]', where *Env* is the type of environment, *l* is a string label used to refer to this environment, *v* lists the universal variables in *form*, *C* is a formula expressing a condition on *v*, *r* specifies ranges for bound variables in *form*, and finally *form* is a single formula or a sequence of formulae. After evaluating a formal text block in a *Theorema* session, it can be referred to (e.g. in a call to a prover) by '*Env*[*l*]'. Concretely, e.g. in (WC), the 'any$[\pi, n]$' makes the definition applicable for all π and all n and the 'bound[allocation$_n[x]$]' makes the 'for all x' to actually range over all allocations x of length n. The real convenience of the 'bound'-construct will be revealed once we collect several definitions into one 'Definition'-environment, e.g. one definition for the axioms (WC), (WR), and (SR), where one 'bound'-statement would suffice for all three axioms. For more details we refer to [1].

[3] For full formalizations in *Theorema* together with proofs see also `http://www.cs.bham.ac.uk/~mmk/economics/theorema`.

by definition." In theorems and lemmas, of course, we explicitly list all preconditions. Furthermore, we split the definitions into individual environments, although the *Theorema* language would allow to collect several formulae into one environment. We decided to proceed this way because we later want to use the definitions on an individual basis, and the current version of *Theorema* allows to access only whole environments, not single formulae.

3.1 Lemma 1: Representation

The first lemma states that the power of a coalition depends only on the holdings of the members of the coalition, but not on the holdings of the other agents.

Lemma["powerfunction-independent", any$[\pi, n, C, x, y]$,
with[allocation$_n[x] \wedge$ allocation$_n[y] \wedge C \subseteq I[n] \wedge$ powerfunction$[\pi, n]$],
$\underset{i \in C}{\forall} (x_i = y_i) \implies (\pi[C, x] = \pi[C, y])$]

In order to prove this, we call the *Theorema* predicate logic prover, see [2], by

Prove[Lemma["powerfunction-independent"],
using→{Definition["powerfunction"],Definition["WR"], Proposition["ref/as"]},
by → PredicateProver, SearchDepth → 100],

which uses the definition of power function, the axiom (WR), and the following (trivial) property of the partial ordering $\geq$ on real numbers in its knowledge base:[4]

Proposition. ["ref/as", any$[a, b]$, $(a = b) \Leftrightarrow (a \geq b \wedge b \geq a)$].

With these settings the lemma is proved fully automatically. It also generates a proof if it is given not only the axioms it needs for a proof but the full theory. *Theorema* generates a human readable proof of the proof search, which is ten pages long for a proof with the full theory, and five pages for the setting used above. This proof can be automatically tidied to a three page proof, which contains only those steps necessary for the final argument, see Fig. 1 to get an impression of how *Theorema* presents a human readable proof (further information is at `http://www.cs.bham.ac.uk/~mmk/economics/theorema`).

3.2 Lemma 2: Domination

The second Lemma states that when the opposing coalitions consist of one element each and the power function is anonymous then the coalition which wins is the one with bigger holdings. In order to formalize Lemma 2 we need in addition to formalize the win set of a transition from an allocation x to an allocation y (those agents which benefit from the transition) and the lose set (those agents to whose detriment the transition is) as well as the notions of domination (x domintates y if the coalition consisting of the win set is, in x, more powerful

[4] This proposition can in turn be proven fully automatically using reflexivity of $\geq$ for the part from left to right and anti-symmetry of $\geq$ for the opposite direction.

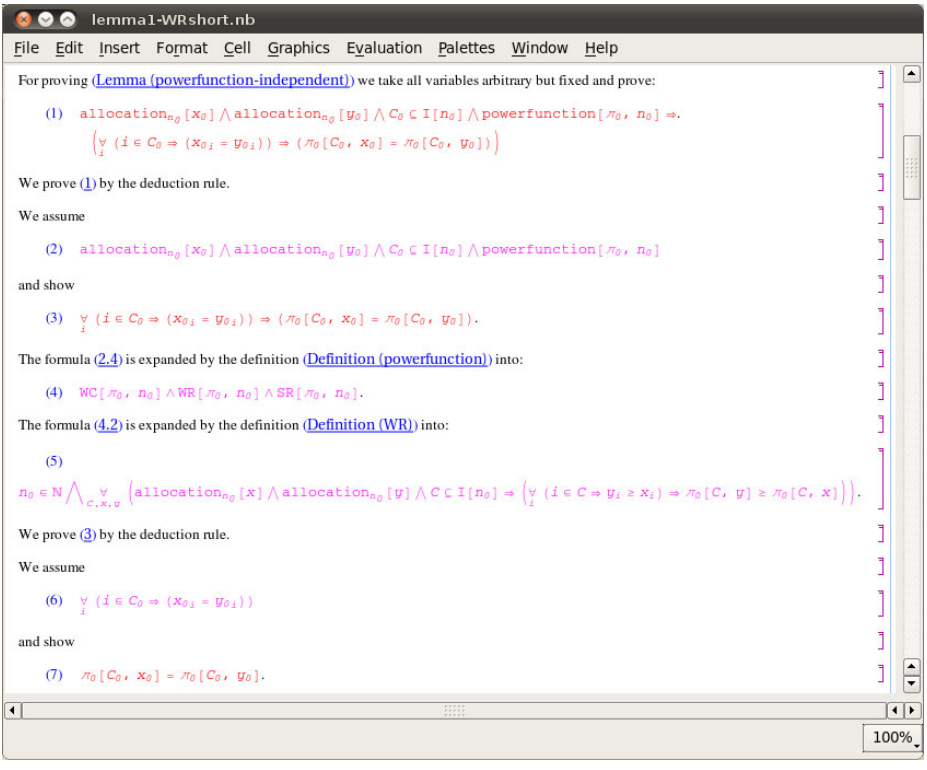


Fig. 1. A human readable proof in *Theorema*

than the coalition consisting of the lose set) and anonymity (invariance under permutations of agents). See also Section 2.

Definition. [“WinLose”, any$[n, x, y]$,
$W[n, x, y] := \{i \mid_{i \in I[n]} y_i > x_i\}$ “W”
$L[n, x, y] := \{i \mid_{i \in I[n]} x_i > y_i\}$ “L”]

Definition. [“domination”, any$[\pi, n, x, y]$,
$\text{dominates}[y, x, \pi, n] :\Leftrightarrow n \in \mathbb{N} \wedge \pi[W[n, x, y], x] > \pi[L[n, x, y], x]$]

Definition. [“anonymity”, any$[\pi, n]$, bound$[\text{allocation}_n[x], \text{allocation}_n[y]]$,

$$\text{anonymous}[\pi, n] :\Leftrightarrow n \in \mathbb{N} \wedge \underset{\substack{\sigma \\ \text{permutation}[\sigma, I[n]]}}{\forall} \underset{\substack{Cx, Cy, x, y \\ Cx \subseteq I[n] \wedge Cy \subseteq I[n]}}{\forall} \underset{i}{\forall} (((i \in Cx) \iff (\sigma[i] \in Cy)) \wedge (x_i = y_{\sigma[i]})) \implies (\pi[Cx, x] = \pi[Cy, y])]$$

Lemma 2 can then be formulated as follows:

Lemma. ["ANdominates", any$[n, x, y]$, with$[n \in \mathbb{N} \wedge n \geq 2 \wedge \text{allocation}_n[x] \wedge \text{allocation}_n[y] \wedge (W[n, x, y] = \{1\}) \wedge (L[n, x, y] = \{2\})]$,

$$\underset{\substack{\pi \\ \text{anonymous}[\pi,n] \wedge \text{powerfunction}[\pi,n]}}{\forall} (\text{dominates}[y, x, \pi, n] \Leftrightarrow x_1 > x_2)\,]$$

While this is apparently intuitively correct, a formal argument is not completely trivial. Since anonymity involves permutations of the agents, a formal proof requires auxiliary knowledge about permutations. In the concrete case, we provide a lemma about permuted tuples, namely that when swapping the first two elements in a tuple the second element in the new tuple is equal to the first of the original:

Lemma. ["perm swap", any$[x]$, perm$[x, \sigma_{1,2}]_2 = x_1$],

where perm$[x, \sigma]$ stands for the tuple x permuted by σ, and $\sigma_{1,2}$ is the permutation swapping the first and second component while leaving the rest unchanged. Furthermore, we use a lemma saying that the power does not change when swapping agents 1 and 2, i.e.

Lemma. ["ANswap", any$[n, \pi, x]$, with$[\text{anonymous}[\pi, n] \wedge \text{allocation}_n[x]]$, $\pi[\{1\}, x] = \pi[\{2\}, \text{perm}[x, \sigma_{1,2}]]$].

Lemma["perm swap"] can be verified without effort based on the definitions only, whereas Lemma["ANswap"] needs the definitions of the concepts involved plus idempotency of swapping, which is trivial but still automatically verified based on the definition of $\sigma_{1,2}$. These proofs, and all that will follow, have been generated by the *Theorema* set theory prover, see [20]. Equipped with Lemma["perm swap"] and Lemma["ANswap"] in the knowledge base, a three page human-understandable proof of the two directions of Lemma 2 is obtained. Note, however that the introduction of the auxiliary lemma is a eureka step, since finding the right permutation is key to the proof of our main lemma; it is difficult to see how *Theorema*—with its currently integrated theorem provers—could automatically find this permutation. This shows that the main idea of the proof requires a fairly intuitive understanding of permutations which needed to be made more explicit when instructing *Theorema*.

The crucial question is, of course, how to obtain appropriate intermediate lemmas. One approach in this kind of *theory exploration* is to always prove all properties of interactions between available concepts before introducing a new concept. This approach is advocated for instance in [3]. In the concrete case, it would require proving many properties of permuted vectors and swaps before talking about anonymity. *Theorema*, however, also supports a much more goal-oriented approach: in the case of a failing proof attempt, it displays the partial proof, allowing a human check of where resources are used, and whether

Theorema can be better guided. In most cases, formulating an appropriate lemma is then a straightforward exercise, so that even this can be automated. There is a literature on lemma speculation from failing proofs (see e.g. [7]); some mechanisms have been implemented in the *Theorema* system as well (see [1]). In this case study, these tools have not been employed.

3.3 Lemma 3: The Core

The *dominion* $D[Y, \pi, n]$ denotes the set of all allocations that are dominated by an allocation in Y. Using this, the *core*, i.e. the set of undominated allocations, can simply be defined as $K[\pi, n] := X[n] \setminus D[X[n], \pi, n]$. The third lemma specializes to the case of three agents and consists of two parts. Firstly, if the core is empty then the tyrannical elements (one agent possesses everything) are dominated by the half splits (e.g., $\langle 1/2, 1/2, 0\rangle$ dominates $\langle 0, 0, 1\rangle$):

Lemma. ["Core,n=3,a", any[π],
$(K[\pi, 3] = \emptyset) \implies \underset{i,j,k \in I[3]}{\forall} (\text{distinct}[i, j, k] \implies t[i, 3] \in D[\{s[j, k, 3]\}, \pi, 3])$].

On the other hand, for anonymous power functions, the half way splits are never dominated by the tyrannical elements (e.g., $\langle 0, 0, 1\rangle$ does not dominate $\langle 1/2, 1/2, 0\rangle$), written in *Theorema* as follows:

Lemma. ["Core,n=3,b", any[π], with[anonymous[$\pi, 3$] $\wedge$ powerfunction[$\pi, 3$]],
$\underset{i,j,k \in I[3]}{\forall} (\text{distinct}[i, j, k] \implies s[j, k, 3] \notin D[\{t[i, 3]\}, \pi, 3])$].

It is clear by the definitions of D and domination, that the win and lose sets under tyrannical elements and the half splits, e.g. $W[3, t[i, 3], s[j, k, 3]]$, will play an essential role in the proofs. We then use the computational capabilities of *Theorema* in order to get some intuition about these entities. Using the built-in computational semantics of the *Theorema* language, one can do some experiments like computing $W[3, t[1, 3], s[2, 3, 3]]$ and $L[3, t[1, 3], s[2, 3, 3]]$ resulting in $\{2, 3\}$ and $\{1\}$, respectively. After some calculations of this kind, the following generalization can be conjectured as a lemma:

Lemma. ["s dominates t", any[$i, j, k \in I[3]$], with[distinct[i, j, k]],
$W[3, t[i, 3], s[j, k, 3]] = I[3] \setminus \{i\}$
$L[3, t[i, 3], s[j, k, 3]] = \{i\}$],

The proof of this Lemma can be done by pure computation, since the universal quantifier over i, j, k boils down to just testing finitely many cases, namely the six distinct choices of $i, j, k \in I[3]$. The neat integration of proving and computing in the *Theorema* system is of great value in this kind of investigation, because the statements need no reformulation when switching between proving and computing.

For proving the first part of Lemma 3, instead of going back to the definition of the core, we use a theorem of Jordan that connects the core and the power

of tyrannical allocations, see [8]. Together with Lemma["s dominates t"], this proof goes through without further complications. For the second part, *Theorema* comes up with an indirect proof using a variant of Lemma["s dominates t"] with the roles of s and t interchanged and specialized versions of the axioms (WC) and (SR) and of Lemma["ANswap"] (introduced in the context of Lemma 2), for the case $n = 3$.

3.4 Pseudo-Code and Its Computational Content

The main result of [10] is a classification of the possibilities which a stable set can have in three agent pillage games with continuous, responsive, and anonymous power functions. No stable set may exist but—if one does—it may have up to 15 elements. How these elements are determined (in dependency of π) can be summarized in form of some pseudo-code. This pseudo-code can be represented as an algorithm in *Theorema* as shown in Fig. 2.

Algorithm["StableSet2", any[π],

$$
\begin{array}{l}
\text{stableSet}[\pi] := \\
\left\{\begin{array}{l}
\left\{\begin{array}{ll}
\text{"no stable"} & \Leftarrow \text{empty}[R[1,\pi]] \\
\text{where}[\mathcal{S} = \text{dyadicSet}[0,3] \cup \bigcup\limits_{i=1,\ldots,3} S[i,\pi], & \\
\left\{\begin{array}{ll}
\mathcal{S} \cup P[\pi] & \Leftarrow \neg\text{fullSet}[\mathcal{S} \cup D[\mathcal{S},\pi,3]] \\
\mathcal{S} & \Leftarrow \text{fullSet}[\mathcal{S} \cup D[\mathcal{S},\pi,3]] \\
\text{"unknown X"} & \Leftarrow \text{otherwise}
\end{array}\right. & \Leftarrow \neg\text{empty}[R[1,\pi]] \Leftarrow (*) \\
\text{"unknown R"} & \Leftarrow \text{otherwise}
\end{array}\right. \\
\text{dyadicSet}[1,3] \backslash \text{dyadicSet}[0,3] \qquad \Leftarrow \text{otherwise}
\end{array}\right.
\end{array}
$$

with $(*)$ to be replaced by $\pi[\{1\}, t[1,3]] \geq \pi[\{2,3\}, t[1,3]]$.

Fig. 2. Algorithm to compute a stable set written in *Theorema* notation

The algorithm in Fig. 2 makes use of several sets and conditions which we explain only partly in the following, since not all details are of importance in this context. Important is that certain sets (such as dyadicSet$[1,3]$) are computational, whereas others (such as $R[1,\pi]$) are not. For details of these constructs see [10].

This algorithm is non computational in several ways. For $n = 3$, however, it is a significant improvement on the Roth-Jordan [9] algorithm to determine stable sets, since the latter is non-computational in even more ways.[5] In this section we will discuss how the algorithm, which is written in the first place to summarize

[5] First, the Roth-Jordan algorithm starts with the core without giving an effective means for computing it. Second, for an empty core it provides no clue for finding an initial iterate. Third, the iteration step is not computational since it involves the computation of undominated sets, which are typically infinite. Finally, it is not clear whether terminating at a set $\mathcal{S}$ which is not externally stable means that no stable set exists, or merely that further steps must be taken independently of the algorithm. For details, see [9] or [10].

the results in a concise form, can be used to determine stable sets by a mixture of proving and computing.

We have tested the implementation for the three specific power functions introduced by Jordan [8], *strength in numbers* (SIN), *Cobb-Douglas* (CD), and *wealth is power* (WIP).

If a concrete power function π is given, the algorithm has first to test condition $(*)$. Since we assume that π is computable, both $\pi[\{1\}, t[1,3]] \in [0,1]$ and $\pi[\{2,3\}, t[1,3]] \in [0,1]$, hence $\pi[\{1\}, t[1,3]] \geq \pi[\{2,3\}, t[1,3]]$ can be decided. If the condition is false the stable set is given by the last line of the algorithm.[6] It is computed by *Theorema* as the set $\{\langle 1/2, 1/2, 0\rangle, \langle 1/2, 0, 1/2\rangle, \langle 0, 1/2, 1/2\rangle\}$. This case was concretely tested for the 'strength in numbers' power function, defined as

$$\mathrm{SIN}\pi_\nu[C,x] := \sum_{i\in C} (x_i + \nu)$$

(with $\nu > 1$) for the concrete value of $\nu = 2$.

If the condition is true, however, the algorithm must check whether a particular set $R[1,\pi]$ is empty or not. For finite $R[1,\pi]$, the ad-hoc method to test $R[1,\pi] = \emptyset$ is to compute $R[1,\pi]$ (by enumerating its elements) and then comparing it to the empty set $\emptyset$, which can all be done in a *Theorema* computation. Unfortunately, the set $R[1,\pi]$ is defined in terms of the set $M[1,\pi]$, which in turn is defined in form of the set $B[1,\pi]$ of all triples, where agent 1 on its own is equally powerful as agents 2 and 3 together, i.e.

$$B[1,\pi] := \{x \in X[3] \mid \pi[\{1\}, x] = \pi[I[3]\backslash\{1\}, x]\}\,.$$

The set $M[1,\pi]$ is then a subset of $B[1,\pi]$ such that the x_1 are maximal, and $R[1,\pi]$ those elements in $M[1,\pi]$ which are maximal from the viewpoint of agents 2 and 3. Since $B[1,\pi]$ is typically infinite, testing $R[1,\pi] = \emptyset$ by computation as described above would fail. Without any further knowledge on $R[1,\pi]$, the algorithm returns "unknown R", which indicates that it has insufficient knowledge on the set R and for this reason insufficient knowledge to determine the stable set.

If, however, for the concrete power function π there is additional knowledge that allows us to decide $R[1,\pi] = \emptyset$ (without actually computing $R[1,\pi]$), the algorithm may make use of this knowledge and continue. Concretely in the algorithm above, we provide the following lemma:
Lemma["emptyR,Cobb Douglas", any[ν], empty[$R[1, \mathrm{CD}\pi_\nu]$]],
where $\mathrm{CD}\pi_\nu$ is the 'Cobb-Douglas' power function defined as

$$\mathrm{CD}\pi_\nu[C,x] := |C|^\nu \Big(\sum_{i\in C} x_i\Big)^{1-\nu} \quad \text{for } 0 \leq \nu \leq 1.$$

[6] The allocations in which each agent has either nothing or $\left(\frac{1}{2}\right)^j$ for some integer $j \leq i$ form a so-called *dyadic set*, represented as dyadicSet$[i,n]$ (for n agents). For three agents the previously mentioned tyrannic allocations and half-splits as well as the ones of type $\langle 1/2, 1/4, 1/4\rangle$ play an important role in determining the stable set.

In this case, according to the algorithm, no stable set exists. Note, however, that the knowledge must be formulated appropriately. For instance, in the algorithm above it was not possible to use the usual *Theorema* notation $R[1,\pi] \neq \emptyset$ instead of $\neg\text{empty}[R[1,\pi]]$: if $R[1,\pi] \neq \emptyset$ was given as an auxiliary lemma, the computation engine would have had to process negated equalities in an appropriate manner, which the current version of *Theorema* is not capable of.

If $R[1,\pi]$ is known not to be empty, further knowledge is necessary. Most importantly, does the current iteration of the computed candidate stable set together, with the allocations dominated by it, include all possible allocations? Formally, is $\text{fullSet}[\mathcal{S} \cup D[\mathcal{S},\pi,3]]$ true or false? This will typically not be computational, since $D[\mathcal{S},\pi,3]$ is infinite. Hence for any particular given power function π a corresponding lemma is necessary, which states whether the property holds or not. In case of the 'wealth is power' power function, defined as

$$\text{WIP}\pi[C,x] := \sum_{i \in C} x_i,$$

the property does not hold since there are three points which are not dominated by the allocations computed so far. In this case, the stable set is computed by *Theorema* to $\mathcal{S} \cup P[\pi]$, which is evaluated directly, since $P[\pi]$ is a set of at most three elements explicitly defined in [10].

The algorithm presented above does not check whether a power function satisfies the additional axioms (continuity, responsiveness, and anonymity), or even whether the functions supplied are actually power functions (satisfying axioms WC, WR, and SR). As a consequence, the algorithm may give wrong answers if applied inappropriately. This can be remedied by adding further conditions to the functions. While this makes the application safer on the one hand, it increases the proof obligations on the other hand.

Note that when applying the algorithm to concrete examples, part of the knowledge may typically be computed, certain information about sets derived from $R[1,\pi]$ must be given in form of lemmas. That is, the algorithm consists of a mixture of proving and computing. It is conceivable that *Theorema* could be extended to make use of the underlying *Mathematica* system to compute the sets $B[i,\pi]$ for particular power functions π.

4 Added Value—Price Paid

In this section we summarize the added value from the point of view of a researcher in theoretical economics. The added value is partly due to the formalization effort and could have been achieved with any formal system, partly, however, it is specifically due to *Theorema* and its features. Furthermore we discuss the effort necessary to do such a formalization.

An obvious point to mention is the greater precision that a formal system requires. This is an advantage (enhanced clarity, greater reliability) and at times a disadvantage (greater effort) at the same time. A concrete example where the formal precision played a role was the characterization of the core as

$K = \{x \in X \,|x_i > 0 \Leftrightarrow \pi(\{i\}, x) \geq \pi(I \setminus \{i\}, x)\}$. In this definition a free index i is used. Two standard interpretations are possible, an existentially quantified one ('there is an i such that'), or a universally quantified one ('for all i holds'). When translating into *Theorema* first the existentially quantified translation was chosen. However, this was later found to be the wrong one when it was used.

Another obvious advantage is that we can have much higher confidence in lemmas and theorems which are formally proved. In addition, the system clearly states all the knowledge used for proving a particular assertion and any hidden assumptions have to be made explicit. On the other hand, an automated theorem prover will typically be inundated with too much information so that the knowledge used to prove an assertion is typically minimal. This is useful knowledge since it allows us to generalize statements.

There are also particular advantages of the computational aspects of *Theorema*. They allow computation and checking particular structures for specific examples. For instance, for a given power function π it is possible to compute particular sets (such as $R[i, \pi]$ or $P[\pi]$) and to see whether these correspond to the intuition. If they do, this gives confidence that the formalization accurately mirrors the intuition. If they do not, then either the intuition needs to be changed or the formalization does not reflect what actually should have been formalized and needs to be changed. Of course, also incorrect statements may be discovered this way. For instance, it led to an adjustment of the algorithm in Fig. 2 in which the last case was incorrectly copied from the corresponding lemma into it. That is, a mistake in the algorithm was detected, although no attempt was made to verify it.[7]

A particular advantage of using *Theorema* is due to the fact that some reasoners—in particular the set theory prover used mostly for our study—use an interface to the computation engine, so that proving and computing are well-integrated in the *Theorema* system. In this case study, this feature turned out to be useful when the whole proof of Lemma["s dominates t"] was shifted to just one simple computation on finite sets. In our concrete application example, we also make use of the computational parts of the algorithm in Fig. 2 to determine the stable set. The algorithm contains algorithmic parts but needs at two steps an oracle which can be given in form of lemmas. These can in turn be formally proved in *Theorema*. The possibility of *Theorema* working in a 'compute' mode makes it relatively painless to combine reasoning and computation. This makes it possible to determine the stable set for concrete power functions by a mixture of reasoning and computation.

As mentioned in the previous section it seems feasible to allow *Theorema* to move more tasks from its reasoning part into its computation part (for specialized power functions). One way to achieve this is to represent infinite sets finitely. More work in this direction is necessary.

Obviously using a system such as *Theorema* has a price. The formalization is more labour intense than formalizing the knowledge just on paper or using

[7] As the algorithm summarizes the central results in [10], we are still in the process of verifying the proofs in paper—informally and formally.

LaTeX. However, just writing down the definitions, lemmas, and theorems in *Theorema* is—after an introduction phase of a day—almost as painless as using LaTeX. Formalizing the knowledge is relatively easy, formally proving the lemmas and theorems, however, is typically labour intense and requires knowledge which cannot be acquired so quickly. Certain formalizations need to be changed in order to avoid certain pitfalls of the reasoners. Additionally it is necessary to know which integrated prover to use best and to adjust it by setting suitable options. This requires expert knowledge. Furthermore, it may be necessary to introduce suitable auxiliary lemmas to prove statements. This makes it currently unlikely that there will be a big uptake in using such systems, although in balance the extra work required—at least for the formal properties proved in this work—does not look too huge with one to two days of work for a formal statement. This effort may go up as the statements get more complicated as the paper proceeds, which is partly due to the fact that typically authors acquire some intuition about the concepts introduced in earlier sections, and this intuition is rarely made explicit by additional lemmas or corollaries. That is, this increased effort could be considered as a price paid by the author and an added value on the side of the reader of a formalized paper. It should be noted that just using *Theorema* for representing knowledge and making use of this knowledge, e.g. by evaluating the algorithm for concrete power functions can uncover mistakes and thereby improve the reliability of the results.

5 Conclusion

The true benefit of a formalization can only be obtained if it is used. Typically you either prove something with it, or you use it in some computation. The latter is only possible in finite domains. In the examples, for instance, it meant that once a set is reduced to a finite set of concrete values, lemmas need not be formally proved as their truth status can be shown by exhaustive computation of all possible cases. This dual feature of computation and proving also allowed us to use the algorithm for computing the stable set for power functions for which certain features had been established (or claimed) by lemmas. Note that *Theorema* does not insist on proving assertions claimed. This makes it very easy to cite external theorems (concretely, in this example, theorems established by Jordan previously) without reproving them. While this is very convenient, it has a downside, namely it is easy to base a theorem on a lemma which is not proved or is even wrong.

A very useful feature of *Theorema* is the possibility to generate incomplete proofs. This allows both detection of potential problems with the formalizations (e.g. inconsistent argument orderings) and monitoring of whether the integrated *Theorema* prover used is making useful steps towards a solution. Moreover, the study of an incomplete proof typically helps greatly in the formulation of lemmas that then help the prover to succeed in a subsequent run, which is quite helpful since the integrated provers of *Theorema* are typically not interactive. This means, that, if they do not find a proof fully automatically, then the only

possibility to guide the search is to adjust suitable options (which requires good knowledge of the inner workings of such a prover), or to introduce additional auxiliary lemmas.

As with any other theorem proving system, applying *Theorema* requires good knowledge of the system. On the positive side, the *Theorema* input language is very flexible and allows—after a very brief introduction to the system—for a natural representation which is close to a paper representation. Care should, however, be taken, since writing down statements in *Theorema* does not mean that they are correct, or can be directly used in the form they have been inputted. The input language is untyped which makes it easy to write things down. However, it also means that it is easy to introduce mistakes, e.g., *Theorema* would not complain if you defined a power function with the argument order as in powerfunction$[\pi, n]$ and later used it in form of powerfunction$[n, \pi]$. However, proofs may not be established any more in such a case.

While proving assertions formally using the *Theorema* system requires typically good knowledge of the proof in the first place, the formalization effort pays off in several ways. Above all it is possible to gain greater confidence in the correctness of the assertions, and it is possible to make experiments—concretely with specific instances of power functions—which otherwise are labour intense and error-prone.

References

1. Buchberger, B., Dupre, C., Jebelean, T., Kriftner, F., Nakagawa, K., Vasaru, D., Windsteiger, W.: The Theorema Project: A Progress Report. In: Kerber, M., Kohlhase, M. (eds.) Proceedings of CALCULEMUS 2000, Symposium on the Integration of Symbolic Computation and Mechanized Reasoning, pp. 98–113. A.K. Peters, Natick (2000)
2. Buchberger, B., Craciun, A., Jebelean, T., Kovacs, L., Kutsia, T., Nakagawa, K., Piroi, F., Popov, N., Robu, J., Rosenkranz, M., Windsteiger, W.: Theorema: Towards Computer-Aided Mathematical Theory Exploration. Journal of Applied Logic 4(4), 470–504 (2006)
3. Buchberger, B.: Theory Exploration Versus Theorem Proving. In: Proceedings of the Calculemus 1999 Workshop. Electronic Notes in Theoretical Computer Science, vol. 23(3). Elsevier, Amsterdam (1999)
4. de Clippel, G., Serrano, R.: Bargaining, coalitions and externalities: A comment on Maskin, http://ssrn.com/abstract=1304712
5. Deng, X., Papadimitriou, C.H.: On the complexity of cooperative solution concepts. Mathematics of Operations Research 19(2), 257–266 (1994)
6. Grandi, U., Endriss, U.: First-Order Logic Formalisation of Arrow's Theorem. In: He, X., Horty, J., Pacuit, E. (eds.) LORI 2009. LNCS, vol. 5834, pp. 133–146. Springer, Heidelberg (2009)
7. Ireland, A., Bundy, A.: Productive Use of Failure in Inductive Proof. Journal of Automated Reasoning 16(1), 79–111 (1995)
8. Jordan, J.S.: Pillage and property. Journal of Economic Theory 131(1), 26–44 (2006)
9. Jordan, J.S., Obadia, D.: Stable Sets in Majority Pillage Games, mimeo (November 2004)

10. Kerber, M., Rowat, C.: Stable Sets in Three Agent Pillage Games, Department of Economics Discussion Paper, University of Birmingham, 09-07 (June 2009), http://papers.ssrn.com/sol3/papers.cfm?abstract_id=1429326
11. Lakatos, I.: Proofs and Refutations. Cambridge University Press, Cambridge (1976)
12. Lucas, W.F.: A game with no solution. Bulletin of the American Mathematical Society 74(2), 237–239 (1968)
13. Maskin, E.: Bargaining, Coalitions and Externalities. mimeo (June 2003)
14. McKelvey, R.D., McLennan, A.M., Turocy, T.L.: Gambit: Software Tools for Game Theory, Version 0.2010.09.01. mimeo (2010), http://www.gambit-project.org
15. Nipkow, T.: Social Choice Theory in HOL: Arrow and Gibbard-Satterthwaite. Journal of Automated Reasoning 43, 289–304 (2009)
16. Vestergaard, R., Lescanne, P., Ono, H.: The Inductive and Modal Proof Theory of Aumann's Theorem on Rationality. mimeo (2006)
17. von Neumann, J., Morgenstern, O.: Theory of Games and Economic Behavior, 1st edn. Princeton University Press, Princeton (1944)
18. Wiedijk, F.: Formalizing Arrow's Theorem. Sādhanā 34(1), 193–220 (2009)
19. Windsteiger, W., Buchberger, B., Rosenkranz, M.: Theorema. In: Wiedijk, F. (ed.) The Seventeen Provers of the World. LNCS (LNAI), vol. 3600, pp. 96–107. Springer, Heidelberg (2006)
20. Windsteiger, W.: An Automated Prover for Zermelo-Fraenkel Set Theory in Theorema. JSC 41(3-4), 435–470 (2006)

View of Computer Algebra Data from Coq

Vladimir Komendantsky, Alexander Konovalov, and Steve Linton

School of Computer Science,
University of St Andrews,
St Andrews, KY16 9SX, UK
{vk,alexk,sal}@cs.st-andrews.ac.uk

Abstract. Data representation is an important aspect of software composition. It is often the case that different software components are programmed to represent data in the ways which are the most appropriate for their problem domains. Sometimes, converting data from one representation to another is a non-trivial task. This is the case with computer algebra systems and type-theory based interactive theorem provers such as Coq. We provide some custom instrumentation inside Coq to support a computer algebra system (CAS) communication protocol known as SCSCP. We describe general aspects of viewing OpenMath terms produced by a CAS in the calculus of Coq, as well as viewing pure Coq terms in a simpler type system that is behind OpenMath.

1 Introduction

In this paper, we describe a realistic example of communication between a state-of-art interactive prover Coq [23] and a modern computer algebra system GAP (Groups, Algorithms, Programs) [11]. What makes this communication most interesting is the complexity of both systems. Indeed, Coq is a constructive system based on a variant of the predicative Calculus of Inductive and Coinductive Constructions (CIC, for short) [23]. Coq has a powerful type system featuring the universe of types, `Type`, but is not Turing-complete and does not allow arbitrary non-terminating computations since any recursive function should be defined over a decreasing measure respecting the well-founded induction principle. Therefore Coq is not a "real-world" programming language since it is not in the custom of a mainstream programmer to provide termination guarantees for every program they write. On the other hand, the mathematical type system of GAP [5] originates from the object-oriented paradigm and is therefore more conventional. Besides, GAP is a real-world imperative programming language where any computation is allowed. There is a type of all objects, `IsObject`, but it is not predicative. Hence, instead of mapping of the type system of GAP to that of Coq, we consider a two-stage embedding: first, a partial embedding of OpenMath terms (which are presented in XML) to a simple inductive type `OM` of OpenMath object, and second, a translation of simply typed OpenMath objects to dependently typed objects of the mathematical hierarchy of Ssreflect [13,12]. Already the first stage of our embedding is forgetful in GAP types and disregards

J.H. Davenport et al. (Eds.): Calculemus/MKM 2011, LNAI 6824, pp. 74–89, 2011.

GAP type system as a result of exporting mathematical objects from GAP by means of the standard communication interface SCSCP [22,18].

SCSCP (*Symbolic Computation Software Composability Protocol*, [22]) is an interface that may be used for communications between various computer algebra systems or other compatible software. It has been developed as part of the project SCIEnce (Symbolic Computation Infrastructure in Europe) which is a major 5-year project that brings together CAS developers and experts in computational algebra, OpenMath, and parallel computations. The interface aims to provide an easy, robust and reliable way for users to create and consume services implemented in any compatible systems, ranging from generic services (e.g., evaluation of a string query or an OpenMath object) to specialised (e.g., lookup in the database; executing certain procedure). The interface is lightweight and based on the XML container format in which both data and instructions are represented as OpenMath objects. It is a way of marshalling abstract syntax trees corresponding to mathematical objects. OpenMath [20] is a structure-oriented standard of twelve language elements (integers, doubles, variables, applications etc.). It does not have a natural evaluation semantics, however, semantics can be provided externally from a CAS by mapping *symbols* which are defined in *content dictionaries* (CDs) to computational procedures. OpenMath has renderings in several different encodings, with the most commonly used one being the XML.

SCSCP is now implemented in several computer algebra systems, including GAP [18], KANT, Macaulay2, Maple, MuPAD, TRIP (see [9,10] for details) and has APIs making it easy to add SCSCP interface to more systems. We reported on earlier experiments with adding SCSCP support to Coq in [17].

Motivation. The motivation behind this work is a uniform method to view computer algebra data when proving a theorem in theorem proving assistant. Rather than writing certified decision procedures for computer algebra computation, we follow a different approach where a CAS may be called by the theorem prover user from a proof script as an external hint engine (an oracle, so to speak). This allows to explore large libraries of non-certified computational algorithms available in CASs and yet leaves a possibility to render results of computation as terms of constructive type theory. To make a computation request to a CAS concerning some proof theoretic objects, we have to remove dependent arguments and proofs from these objects and form a simply typed term in a container format that is used for communication between CASs. If the computation is successful, its result is retrieved from such a simply typed term by lifting it to a dependently typed term and supplying the missing proof arguments.

Contribution. Our main contribution is the communication plugin for Coq that automates routine data transmission tasks while leaving the theorem proving for the user. We also provide a high-level automation for data conversion between dependently typed and simply typed representations of Coq and CAS respectively. This high-level automation is the view mechanism that is written in Coq, which allows custom extension without the need to recompile the communication plugin. We also provide a method to organise views in a type-theoretic

hierarchy of packed classes [12], which allows having different views of the same type of mathematical object. Which exact view is relevant in a given situation can be determined by Coq by unification with implicit coercions and canonical structures [21].

Outline. In Section 2, we describe our approach to views of computer algebra data. In Sections 3-5 we discuss a concrete example of communication between Coq and GAP. An example showing control elements of SCSCP is given in Section 6. In Section 7 we discuss related work and GAP type system and why we chose not to refer to GAP types in our approach. Finally, in Section 8 we give our conclusions and propose future directions.

2 Views of Computer Algebra Data

Views in the sense of Wadler [24] are a method to define correspondence between any, possibly abstract, unstructured data type and an inductive data type with structure. By viewing an abstract type as a free inductive type we allow ourselves to pattern-match on objects of the abstract data type. For example, we may view built-in signed or unsigned integers as Peano numbers, and so on. We may also view a structured data type as some other structured data type, for example, if we need to pinpoint some type dependencies in a given simple type.

Viewing abstract types as inductive ones has its clear advantages. In programming languages, data abstraction is a powerful mechanism supporting efficient computation and allowing to generalise over concrete structure. However, abstraction hides representation, and we need to find a representation for the purpose of reasoning. Representing abstract data as data with free inductive structure allows to decompose it in a chosen way by induction. Thus, in a sense, a representation is an access strategy. Specifically for our case, computer algebra data produced by GAP may already have some structure but, in general, this structure does not suffice to construct an object of the corresponding type in Coq because proofs are obviously missing.

It is not our goal to implement general views of any abstract type. Instead, we implement views on an inductive type of OpenMath object. For that, we split view construction in two parts using our communication systems interface implemented as a Coq tactic plugin. First, the computer algebra object is retrieved from GAP by the plugin and parsed as an object of inductive type `OM` of OpenMath terms. Parsing is only partial and may fail if some unexpected language element is encountered. Second, the obtained `OM` object is parsed in Coq and missing proofs are provided to construct the Coq view of the computer algebra object.

Now we define the *base type* of view using a convenient natural-deduction style notation to display inductive type definitions. *Mixins* are introduced as elementary building blocks for containers such as viewType below. A mixin is an inductive type together with projections to its constructors. Below is the base view mixin for a type of OpenMath data called OM:

$$\frac{\mathsf{mixin\_of} : \mathsf{Type} \to \mathsf{Type} \qquad \mathit{viewin} : \mathsf{OM} \to \mathsf{option}\ iT \qquad \mathit{viewout} : iT \to \mathsf{option\ OM}}{\mathsf{ViewMixin}\ \mathit{viewin}\ \mathit{viewout} : \mathsf{viewMixin\_of}\ iT}$$

Here, option is the polymorphic type of fan ordering with values either of the kind Some of a value of a given type, or None, with the latter representing the absence of translation. Field projections are the following:

$$\begin{array}{lcl} \mathsf{viewin} & = & \lambda\ (iT : \mathsf{Type})\ (m : \mathsf{viewMixin\_of}\ iT).\ \mathbf{let}\ (\mathit{viewin},\ \_) := m\ \mathbf{in}\ \mathit{viewin} \\ \mathsf{viewout} & = & \lambda\ (iT : \mathsf{Type})\ (m : \mathsf{viewMixin\_of}\ iT).\ \mathbf{let}\ (\_,\ \mathit{viewout}) := m\ \mathbf{in}\ \mathit{viewout} \end{array}$$

The *class* of the base type is simply its mixin.

$$\mathsf{viewClass\_of} = \mathsf{viewMixin\_of}$$

In the packed type methodology, containers $*$type pack a given representation type with given base classes and the underlying mixin. In the base case, we have only a mixin. The container viewType is given below. The third argument U is required for the purpose of unification.

$$\mathsf{type} : \mathsf{Type}$$

$$\frac{\dfrac{\mathit{viewSort} : \mathsf{Type}}{\mathit{viewClass} : \mathsf{viewClass\_of}\ \mathit{viewSort}} \qquad U : \mathsf{Type}}{\mathsf{ViewPack}\ \mathit{viewSort}\ \mathit{viewClass}\ U : \mathsf{viewType}}$$

Now we have to provide a hint for the unification algorithm that allows to coerce viewType to its representation type. This role is delegated to the field projection viewSort.

$$\mathsf{viewSort} = \lambda\ (t : \mathsf{viewType}).\ \mathbf{let}\ (\mathit{viewSort},\ \_,\ \_) := t\ \mathbf{in}\ \mathit{viewSort}$$

The second field projection, viewClass, is not associated a coercion, which is made on purpose, to prevent multiple coercion chains to the representation type.

$$\begin{array}{lcl} \mathsf{viewClass} & : & \forall\ cT : \mathsf{viewType}.\ \mathsf{viewClass\_of}\ (\mathsf{viewSort}\ cT) \\ \mathsf{viewClass}\ cT & = & \mathbf{let}\ \mathsf{ViewPack}\ \_\ c\ \_\ := cT\ \mathbf{in}\ c \end{array}$$

Finally, we define the constructor ViewType for the type viewType, and input and output view functions:

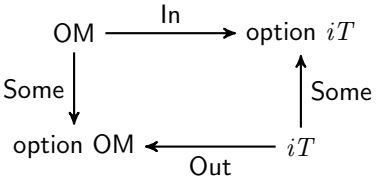
$$\begin{array}{lcl} \mathsf{ViewType}\ T\ m & = & \mathsf{ViewPack}\ T\ m\ T \\ \mathsf{In} & = & \lambda\ (iT : \mathsf{viewType}).\ \mathsf{viewin}\ iT\ (\mathsf{viewClass}\ iT) \\ \mathsf{Out} & = & \lambda\ (iT : \mathsf{viewType}).\ \mathsf{viewout}\ iT\ (\mathsf{viewClass}\ iT) \end{array}$$

Having defined the base type of view we can introduce *higher-level views*. The base type of view can be extended, for example, with a notion of correctness of

a view for the purpose of certification of computer algebra computations, or by adding other kinds of conversion functions for extended usability. The correctness property can be expressed as follows:

$$\forall\ (o : \mathsf{OM})\ (i : iT), \quad \mathsf{In}\ \_\ o = \mathsf{Some}\ iT\ i \quad \leftrightarrow \quad \mathsf{Out}\ \_\ i = \mathsf{Some}\ \mathsf{OM}\ o$$

and illustrated by the diagram below:



Construction of view types given other view types is reminiscent of construction of objects in object-oriented programming. The packed class methodology of Ssreflect [12] helps to define multiple views as a hierarchy of structures. Like in [24], we can have multiple views of the same object. In the following, we discuss views in terms on a concrete example. For the purpose of explaining the basics, we do not yet require a hierarchy of views. Therefore we skip further details of our implementation of higher-level views which is reminicent of the implementation of the mathematical object hierarchy in [12].

3 SCSCP and the Working Example

In our experience, communication with GAP is significantly more structured if SCSCP [22] is used compared to communication by reading off raw output of the GAP interpreter and more straightforward than communication involving direct addressing to internal representation of mathematical objects in GAP. Moreover, communication by SCSCP abstracts over the concrete CAS by employing a common OpenMath abstract syntax tree representation of a mathematical object.

In our working example, we consider computation of the set of roots of a polynomial over a finite field. The standard representation of a polynomial in OpenMath involves the following OpenMath symbols:

- `DMP`, constructor of distributed multivariate polynomials,
- `poly_ring_d`, constructor of polynomial rings,
- `SDMP`, constructor of formal sums,
- `term`, constructor of monomials, such that

  $$\texttt{term}(c, e_1, \ldots, e_n)$$

  represents

  $$c \times x_1^{e_1} \times \cdots \times x_n^{e_n}$$

  where $x_1, \ldots, x_n$ are distinct variable names, and c and $e_1, \ldots, e_n$ are elements of the given polynomial ring.

To represent the following polynomial:

$$x^4y^6 + 3y^5$$

one constructs a syntax tree with the following structure:

```
DMP(poly_ring_d(ℤ, 2), SDMP(term(1, 4, 6), term(3, 0, 5)))
```

Example 1 (Working example). Compute roots of $x^3 - 1$ in the 3-element finite field $GF(3)$.

The abstract syntax tree of the computation request to GAP through SCSCP is the following (disregarding SCSCP control elements that we discuss in Section 6):

```
WS_RootsOfUPol(
  DMP(poly_ring_d(GFp(3), 1),
    SDMP(
      term(times(primitive_element(3), 2), 3),
      term(times(primitive_element(3), 0), 2),
      term(times(primitive_element(3), 0), 1),
      term(times(primitive_element(3), 1), 0))))
```

Provided all the connection prerequisites are met, GAP returns the list of roots:

```
list(power(primitive_element(3), 0),
     power(primitive_element(3), 0),
     power(primitive_element(3), 0))
```

that is, the list containing three occurrences of 2^0 where 2 is the *primitive element*, or, *generator* of the field $GF(3)$.

Next, we show that parsing the GAP response can be defined by the user in Coq. Although mapping of an OpenMath AST to the corresponding CIC object is done by the plugin at the level of Ocaml, the implementation language of Coq. The reason to choose Ocaml for this purpose is that no input-output is possible at the level of pure functional programming in Coq. This mapping is very simple and does not contain any intelligent steps. Our current algorithm is not general enough to account for any possible input OpenMath term. However, even a more general algorithm would not be able to effectively recognise all the possible notational shortcuts used by computer algebra programmers. Therefore a pragmatic approach can be pursued, which we do in Section 4.

4 Term Internalisation

To internalise OpenMath data in Coq, we have to provide a Coq type of OpenMath objects. Despite some discrepancies in interpretation of the OpenMath standard [20] in various computer algebra systems, we have to find some reliable subtype of this possibly intangible type of all OpenMath objects. We start from a very simple inductive type below:

```
Inductive OM : Type :=
  | OMInt : Z -> OM
  | OMVar : string -> OM
  | OMSym : string -> string -> OM
  | OMApp : OM -> seq OM -> OM.
```

where `Z` and `string` are the standard Coq types of integer and string respectively, and `seq` is the type of list used in Ssreflect:

```
Inductive seq (T : Type) : Type :=
  | Nil : seq T
  | Cons : T -> seq T -> seq T.
```

Thus we treat OpenMath integers, variables, symbols (which are in effect pairs of strings) and applications. Some other commonly used kinds of OpenMath object such as attributed objects are not in this inductive type because, for the moment, we restrict ourselves to simple OpenMath data directly related to computation.

Next, we define some notational conventions in Fig. 1 to be used in our example below. Underscores denote the implicit argument that can be inferred by Coq from other arguments or from the type of the expression, that is, in this instance, the type `T` of element given the type of list of `T`.

notation	denotation
`[::]`	`Nil _`
`x :: s`	`Cons _ x s`
`[:: x1]`	`Cons _ x1 (Nil _)`
`[:: x1, x2, .., xn & s]`	`Cons _ x1 (Cons _ x2 .. (Cons _ xn s) ..)`
`#i`	`OMInt i`
`$v`	`OMVar v`
`D$$N`	`OMSym D N`
`(x1 @@ xs)`	`OMApp x1 xs`
`(x1 @ x2)`	`OMApp x1 :: [:: x2]`
`(x1 @ x2 ; .. ; xn)`	`OMApp x1 (x2 :: .. [:: xn] ..)`

Fig. 1. Notational definitions

Further notational shorthands for OpenMath idioms are given in Fig. 2, where `Z_of_nat` is the canonical embedding of natural numbers (type `nat`) into integers (type `Z`). Therefore `OMlist` is the `OM` version of the OpenMath list symbol, and

name	definition
`OMlist`	`"list1"$$"list"`
`GFp n`	`("setname2"$$"GFp" @ #(Z_of_nat n)).`

Fig. 2. Auxiliary definitions

`GFp n` is the `OM` version of the OpenMath object representing the general field of order `n` where `n` is prime. The latter condition of primality is not recorded anywhere either in the OpenMath object or in its `OM` counterpart. In CASs, hypothetical reasoning is not supported and, naturally, there is no need to include proof-related data in objects. We need to recover such data by constructing proofs during internalisation of the OpenMath object.

Given a response containing a list of polynomial roots computed by GAP and transmitted through SCSCP, we can parse the AST using shorthand string tests such as the ones below:

```
Definition is_OMlist o := is_OMsymbol o "list1" "list".
Definition is_OMpower o := is_OMsymbol o "arith1" "power".
Definition is_OMprimelt o :=
  is_OMsymbol o "finfield1" "primitive_element".
```

where `is_OMsymbol` is a simple generic test on pairs of strings, and collect initial simply-typed information from this response using the following function:

```
Definition rootOfUPol (o : OM) : option (nat * Z) :=
  match o with
    | OMApp o os =>
        if is_OMpower o then
          match os with
            | [:: OMApp o1 [:: o2] ; o3] =>
                if is_OMprimelt o1 then
                  match o2, o3 with
                    | #n, #m => Some (Z_to_nat n, m)
                    | _, _ => None
                  end
                else None
            | _ => None
          end
        else None
    | _ => None
  end.
```

The result of this function is an optional pair of numbers to allow for inputs on which our simple data collection algorithm is undefined. If result is defined, we have a pair of a field order and an exponent coefficient. This is the simply-typed data to put in dependently typed mathematical objects of Ssreflect.

We need to define concrete field elements in order to define a polynomial:

```
Definition Fp3_0 : Fp_field 3 prime3 := Ordinal 3 0 (erefl true).
Definition Fp3_1 : Fp_field 3 prime3 := Ordinal 3 1 (erefl true).
Definition Fp3_2 : Fp_field 3 prime3 := Ordinal 3 2 (erefl true).
```

where `prime3` is a proof of the natural number 3 being prime. This proof is required to define the type `Fp_field 3 prime3` which is a Coq representation of the 3-element finite field $GF(3)$. The constructor `Ordinal` takes three arguments: a finite bound (a natural number), a natural number m and a proof that m is less than the bound.

The primitive element of order `n` and exponent with base a primitive element are declared as follows:

```
Definition primelt n n' (nn' : S n = n') (pn' : prime n')
  : Fp_field n' pn' := ...
Definition primelt_exp n n' (nn' : S n = n') (pn' : prime n') (m : Z)
  : Fp_field n' pn' := ...
```

The latter function is defined in terms of the former one. We choose to introduce arguments for each of the primitive element and field order because inferring one from the other can introduce unnecessary dependencies in types.

Internal representation of a polynomial root is constructed by `primelt_exp` from a pairs collected using `rootOfUPol` by a function with the following declaration:

```
Definition rootOfUPol_i :
  (nat * Z) -> {n : nat & forall (pn : prime n), Fp_field n pn} := ...
```

Therefore one should solve a *proof obligation*, that is, provide a primality proof `pn`, for every root to obtain it in the explicit form and not as a product. Since such proof obligations arise in a systematic way, as a result of application of a version of a map functional to `rootOfUPol_i`, it is a relatively straightforward task to automate their generation at the internalisation stage (using the tactic language of Coq).

Putting it all together, the main internalisation function that takes an `OM` object and returns a sequence of roots (which is complicated by primality proof obligations) only if it is defined on the input has the following definition:

```
Definition RootsOfUPol_i (o : OM) :
  option (seq {n : nat & forall (pn : prime n), Fp_field n pn}) :=
  match o with
    | OMApp o os =>
        if is_OMlist o
        then
          match mapo rootOfUPol os with
            | None => None
            | Some rs => mapo (fun s => Some (rootOfUPol_i s)) rs
          end
        else None
    | _ => None
  end.
```

where `mapo` is a stronger version of map functor: it maps a sequence to `None` in case the function, to which `mapo` is applied, yields None if applied to any of the elements of the given sequence. A further comment to the type of `RootsOfUPol_i` is that `n` is in fact fixed in the result in `OM`. However, since this is not explicit anywhere in the result which is a list of arbitrary pairs of numbers, a straightforward way to parse it is to assume `n` can vary. This creates an obligation to prove primality of one and the same `n` for every element of the optional list. Nevertheless, given the optional list constructed by the function `RootsOfUPol_i` and

the observation (a proof) that the first component `n` of every pair in the list is one and the same number, we can construct an optional list of `Fp_field n pn`. The type of the list, this time, depends on `n` and the proof of primality `pn`. In the end, we have to provide only one instance of `pn`, however, for that we need to prove that all the first projections of the list obtained by simple parsing of `OM` data are in fact equal.

5 Term Externalisation

We define a function mapping a natural number, the order of the field, and a sequence of ordinal numbers, the polynomial coefficient sequence, to a sequence of objects of type `OM`. This function externalises the coefficient sequence of a Coq polynomial and builds the corresponding `OM` sequence of simpler structure that can be passed over through SCSCP to GAP:

```
Fixpoint polyseqFp (n : nat) (s : seq 'I_n) : seq OM :=
  match s with
    | [::] => [::]
    | [:: c & s] =>
        [:: ("polyd1"$$"term" @
              ("arith1"$$"times" @
                ("finfield1"$$"primitive_element" @ #(Z_of_nat n));
                #(Z_of_nat c));
              #(Z_of_nat (size s)))
         & polyseqFp s]
  end.
```

Then, we can define

```
Definition myseq := [:: Fp3_2; Fp3_0; Fp3_0; Fp3_1].
```

By standard lazy evaluation of Coq of the term `polyseqFp myseq` we obtain the required sequence. Furthermore, we can construct manually the required polynomial of type commonly used in Ssreflect as follows:

```
Definition mypoly : {poly (Fp_field 3 prime3)} :=
  Polynomial myseq myseq_last.
```

where `myseq_last` is a proof, required by the constructor of formal polynomials, that the last element of the sequence `myseq` is not the field zero.

The Coq representation of the SCSCP request is constructed by the following function taking the field order, the primality proof and a formal polynomial object:

```
Definition RootsOfUPol_e (n : nat) (pn : prime n)
    (p : {poly (Fp_field n pn)}) : OM :=
  ("scscp_transient_1"$$"WS_RootsOfUpol" @
    ("polyd1"$$"DMP" @
      ("polyd1"$$"poly_ring_d" @
        (GFp n); #1);
      ("polyd1"$$"SDMP" @@ polyseqFp p))).
```

The term of the request is obtained by lazy evaluation of the following term in Coq: `RootsOfUPol_e _ _ mypoly`. The normal form obtained by lazy evaluation can be routinely converted into the XML format and sent as an SCSCP request to the GAP server. The server responds with another XML term containing an OpenMath object which is routinely converted to `OM` and internalised according to Section 4. This is a phase in Coq to GAP communication.

6 An Advanced Example

Note that GAP can deal with more complicated constructions than finite fields that are used in the previous example for the sake of simplicity of presentation. The following GAP code constructs a polynomial from the polynomial ring over the algebraic extension over rationals of degree four:

```
x:=Indeterminate(Rationals,"x");
p:=x^4+3*x^2+1;
e:=AlgebraicExtension(Rationals,p);
z:=Indeterminate(e,"z");
w:=z^3-2*z+1;
```

Its roots over this algebraic extension can be calculated using the command `RootsOfPolynomial(e,w)`. The corresponding OpenMath code may be produced using commands `OMPrint(w)` and `OMPrint(RootsOfUPol(e,w))`. Below we provide its more readable POPCORN (Possibly Only Practical Convenient OpenMath Replacement Notation) [15] that shows the SCSCP control elements corresponding to procedure call and procedure result. The user of GAP defines an external name `WS_RootsOfUPol` for the GAP procedure `RootsOfUPol` which they put into a header file accessible to the SCSCP package of GAP. The procedure call is as follows:

```
scscp1.procedure_call( scscp_transient_1.WS_RootsOfUPol( polyd1.DMP(
  polyd1.poly_ring_d_named(field3.field_by_poly(setname1.Q, polyd1.DMP(
    polyd1.poly_ring_d_named(setname1.Q,$x):ref1,
    polyd1.SDMP(polyd1.term(1, 4), polyd1.term(3, 2),
                polyd1.term(1, 0)))), $z),
  polyd1.SDMP(
    polyd1.term(field4.field_by_poly_vector(
      field3.field_by_poly(setname1.Q,
      polyd1.DMP(#ref1, polyd1.SDMP(polyd1.term(1, 4),
      polyd1.term(3, 2), polyd1.term(1, 0)))), [1, 0, 0, 0]), 3),
    polyd1.term(field4.field_by_poly_vector(
      field3.field_by_poly(setname1.Q,
      polyd1.DMP(#ref1, polyd1.SDMP(polyd1.term(1, 4),
      polyd1.term(3, 2), polyd1.term(1, 0)))), [(-2), 0, 0, 0]), 1),
    polyd1.term(field4.field_by_poly_vector(
      field3.field_by_poly(setname1.Q,
      polyd1.DMP(#ref1, polyd1.SDMP(polyd1.term(1, 4),
      polyd1.term(3, 2), polyd1.term(1, 0)))), [1, 0, 0, 0]), 0)))))
```

The result is an application of the "procedure completed" symbol to the list of roots:

```
scscp1.procedure_completed(
[ field4.field_by_poly_vector(field3.field_by_poly(setname1.Q,
    polyd1.DMP(#ref1, polyd1.SDMP(polyd1.term(1, 4),
    polyd1.term(3, 2), polyd1.term(1, 0)))), [1, 0, 0, 0]),
  field4.field_by_poly_vector(field3.field_by_poly(setname1.Q,
    polyd1.DMP(#ref1, polyd1.SDMP(polyd1.term(1, 4),
    polyd1.term(3, 2), polyd1.term(1, 0)))), [1, 0, 1, 0]),
  field4.field_by_poly_vector(field3.field_by_poly(setname1.Q,
    polyd1.DMP(#ref1, polyd1.SDMP(polyd1.term(1, 4),
    polyd1.term(3, 2), polyd1.term(1, 0)))), [(-2), 0, (-1), 0]) ])
```

Structurally, both procedure call and procedure result are quite similar to our working example in Section 3. On the other hand, finding a representation for these in the prover might be a challenge.

7 Related Work

Communication interfaces between proof assistants and CAS were built in [7] and in [8]. The former paper developed an approach where values computed by the CAS-as-an-oracle were used in deductive proofs of primality in the proof assistant. This approach has been further investigated and complemented by an alternative approach in [14] where the values provided by the CAS were rather used in boolean tests, which allowed more efficient proofs by computation in Coq, and so, bigger numbers could be proved to be prime, while still retaining the aspect of correctness-by-proof. The approach in [14] is often referred to as *proof by reflection*.

Two related kinds of development have quite different aims to ours: 1) building a computer algebra system on top of a proof assistant [16,3], and 2) creating a programing environment for development of certified computation [4]. Unlike any of these developments, we do not approach the nature of computational algorithms but rather concern ourselves with the task of representing given computer algebra data in a way acceptable in a theorem prover.

OpenMath, with its structure-oriented notation, is related to a more general case of *abstract syntax tree* (AST) appearing in programming languages research. For example, compare `OM` with the type `aexpr` from [2,19] that gives an AST representation to simple *arithmetical expressions*. This type is instrumental in construction of the *abstract semantics* of the imperative language IMP. For this sake, one has to define semantics of evaluation of instructions to *states*, that is, sets of values for all the variables after execution of the instruction. Evaluation of an AST of an arithmetic or boolean expression yields a value in the Coq type of integers or booleans respectively.

As a result of the comparison with abstract semantics of IMP, we see that OpenMath does not in fact provide a semantics to mathematical expressions but rather employs XML and other container formats to encode ASTs. Indeed, the type `OM` is close to `aexpr` apart from the constructor `OMApp` which has a list argument. There is no analogue of evaluation of arithmetic expressions in the

OpenMath standard. Semantics of mathematical objects represented by OpenMath ASTs, their evaluation, should be defined outside the OpenMath standard, e.g., in a CAS or a prover.

XML technology that is employed by SCSCP is also adopted in the HELM project [1] for management of large open repositories of mathematical knowledge. The methodology is based on a few basic rules, which is reminiscent of the initial motivation behind OpenMath. HELM goes further in that it defines a new distributed collaboration scenario involving various systems of computer mathematics, without a central authority.

In [6], a method is developed to create interactive mathematical documents based on Coq scripts. The method employs a unidirectional Coq to OpenMath encoder in which the OpenMath primitive `OMBIND` is used to denote variable bindings for λ- and Π-abstraction. Although this is probably the best certified encoding as long as OpenMath is concerned, in our development we cannot rely on `OMBIND` since we perform a bidirectional encoding, and converting from a CAS presentation that in general has no bindings to a presentation with bindings seems to be a redundant step.

Discussion of GAP types. GAP [11,5] is a typed programming language in the following sense. Although GAP allows untyped programming elements such as unityped lists, every algebraic object of GAP has type. The *GAP type* of an object is the run-time information about it which has been computed and stored. GAP types consist of a *filter* and a *family* which are described below.

A GAP filter can be defined as a three-valued run-time property: At a given point of execution, the property is either known to GAP and is then a boolean value or unknown. Such values are either set at the time of GAP object creation or computed at run time by special unary GAP functions mapping GAP objects to boolean values, with the output depending on whether or not the object lies in the set associated with the filter. Thus such special functions are essentially characteristic predicates. GAP filters form a lower semilattice whose zero, the GAP filter of all GAP objects, is called `IsObject`. GAP filters are used for method selection in the object-oriented procedure call scheme.

The GAP family is the relation of the object to other objects. Families form a subset of GAP objects such that the following two conditions hold: 1) objects that are (extensionally) equal lie in the same family; and 2) the family of the result of an operation depends only on the families of its operands.

Consider an example application of GAP families. A polynomial ring and its coefficients ring lie in different families, hence the coefficient ring cannot be embedded into the polynomial ring as a subset. Nevertheless we can, for instance, multiply an integer by a polynomial over the ring of integers. The relation between the arguments, namely that the former is a coefficient and the latter a polynomial, is given by the relation between their families.

There is an implementation feature [11], which does not *usually* interact with computations, in the presentation of the family hierarchy which seems to collapse all higher-order families into the family of families. While, in a system with reasoning capabilities such as a theorem prover, this would have led to Russell's

paradox of naive set theory, computer algebra algorithms are not subject to the paradox because they never perform hypothetic reasoning. This nevertheless means that one should avoid simple embedding of the GAP type system into higher-order type theory such as the one of Coq. Instead, we consider taking a suitably rich type-theoretic hierarchy of mathematical structures, Ssreflect [12], and encode GAP objects as objects of type theory. We call such encoding an *internalisation* of the mathematical object.

8 Conclusions and Possible Directions

In Section 2 we discussed our approach to organisation of views of computer algebra data in a consistent hierarchy where multiple inheritance is allowed. At the moment we are using just a few basic views for OpenMath primitives and symbols. Future work should be dedicated to extending this hierarchy. This can be performed along with more case studies in various areas of computer algebra for which there is relevant support in the theorem prover.

In Section 4 we described our approach to OpenMath term internalisation in Coq which employs the type `OM` representing a subset of the OpenMath standard. We showed that parsing of abstract syntax trees can be done purely in Coq, for which we have to provide some minimal input-output instrumentation at the level of the implementation language of Coq, Ocaml. This instrumentation has been routinely used in experiments of socket-based communication between Coq and GAP running an SCSCP server package and is be available from the first author's webpage[1]. The parsing function can in principle be extended to handle generic OpenMath data respecting the type `OM` which in turn can be extended to account for more OpenMath primitives.

The process of inverse translation from Coq to GAP, which is also called mathematical object externalisation, is described in Section 5. Externalisation is simpler than internalisation as long as types are concerned because it forgets dependently-typed parameters. There is an interesting problem of automatising construction of ASTs of type `OM`, possibly by providing a contextual mapping from Coq types to OpenMath symbols.

In Section 7 we mentioned an analogy involving OpenMath and the abstract semantics of arithmetical expressions. Provided this analogy, one might investigate into formal evaluation of OpenMath in Coq in the manner of abstract evaluation as in abstract semantics of the imperative language IMP. Abstract evaluation can target the mathematical hierarchy of Ssreflect and therefore can be a more generic approach to mathematical object internalisation in Coq.

Acknowledgements. The authors are grateful to Georges Gonthier for his kind support of this effort at an earlier stage, and to our referees. This research is part-funded by the EU FP6 project SCIEnce and the Marie Curie IEF fellowship SImPL.

[1] `http://www.cs.st-andrews.ac.uk/~vk/Coq+GAP/`

References

1. Asperti, A., Padovani, L., Sacerdoti Coen, C., Schena, I.: Mathematical knowledge management in HELM. Annals of Mathematics and Artificial Intelligence 38, 27–46 (2003)
2. Bertot, Y.: Structural abstract interpretation: A formal study using Coq. In: Bove, A., Barbosa, L.S., Pardo, A., Pinto, J.S. (eds.) Language Engineering and Rigorous Software Development. LNCS, vol. 5520, pp. 153–194. Springer, Heidelberg (2009)
3. Bertot, Y., Guilhot, F., Mahboubi, A.: A formal study of Bernstein coefficients and polynomials. Mathematical Structures in Computer Science (2011)
4. Boulmé, S., Hardin, T., Hirschkoff, D., Ménissier-Morain, V., Rioboo, R.: On the way to certify computer algebra systems. Electronic Notes in Theoretical Computer Science 23(3), 370–385 (1999); CALCULEMUS 1999, Systems for Integrated Computation and Deduction (associated to FLoC 1999, the 1999 Federated Logic Conference)
5. Breuer, T., Linton, S.: The GAP 4 type system: organising algebraic algorithms. In: ISSAC 1998: Proceedings of the 1998 international symposium on Symbolic and algebraic computation, pp. 38–45. ACM, New York (1998)
6. Caprotti, O., Geuvers, H., Oostdijk, M.: Certified and portable mathematical documents from formal contexts. In: Electronic Proceedings of the 1st International Workshop on Mathematical Knowledge Management, MKM 2001. RISC, Schloss Hagenberg, Austria (2001)
7. Caprotti, O., Oostdijk, M.: Formal and efficient primality proofs by use of computer algebra oracles. J. Symbolic Computation 32(1/2), 55–70 (2001)
8. Delahaye, D., Mayero, M.: Quantifier elimination over algebraically closed fields in a proof assistant using a computer algebra system. In: Proc. Calculemus 2005. ENTCS, vol. 151, pp. 57–73 (2006)
9. Freundt, S., Horn, P., Konovalov, A., Lesseni, S., Linton, S., Roozemond, D.: OpenMath in SCIEnce: Evolving of symbolic computation interaction. In: Davenport, J.H. (ed.) 22nd OpenMath Workshop (July 2009)
10. Freundt, S., Horn, P., Konovalov, A., Linton, S., Roozemond, D.: Symbolic computation software composability. In: Autexier, S., Campbell, J., Rubio, J., Sorge, V., Suzuki, M., Wiedijk, F. (eds.) AISC 2008, Calculemus 2008, and MKM 2008. LNCS (LNAI), vol. 5144, pp. 285–295. Springer, Heidelberg (2008)
11. The GAP Group. GAP – Groups, Algorithms, and Programming, Version 4.4.12 (2008), http://www.gap-system.org
12. Garillot, F., Gonthier, G., Mahboubi, A., Rideau, L.: Packaging mathematical structures. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 327–342. Springer, Heidelberg (2009)
13. Gonthier, G., Mahboubi, A., Tassi, E.: A Small Scale Reflection Extension for the Coq system. Research Report RR-6455, INRIA (2011)
14. Grégoire, B., Théry, L., Werner, B.: A computational approach to pocklington certificates in type theory. In: Hagiya, M. (ed.) FLOPS 2006. LNCS, vol. 3945, pp. 97–113. Springer, Heidelberg (2006)
15. Horn, P., Roozemond, D.: OpenMath in SCIEnce: SCSCP and POPCORN. In: Carette, J., Dixon, L., Coen, C.S., Watt, S.M. (eds.) MKM 2009, Held as Part of CICM 2009. LNCS, vol. 5625, pp. 474–479. Springer, Heidelberg (2009)
16. Kaliszyk, C., Wiedijk, F.: Certified computer algebra on top of an interactive theorem prover. In: Kauers, M., Kerber, M., Miner, R., Windsteiger, W. (eds.) MKM/CALCULEMUS 2007. LNCS (LNAI), vol. 4573, pp. 94–105. Springer, Heidelberg (2007)

17. Komendantsky, V., Konovalov, A., Linton, S.: Interfacing Coq + Ssreflect with GAP. In: Proc. User Interfaces for Theorem Provers (UITP) 2010. ENTCS, Elsevier, Amsterdam (to appear 2010)
18. Konovalov, A., Linton, S.: SCSCP – Symbolic Computation Software Composability Protocol, Version 1.2, GAP package (2010), http://www.cs.st-andrews.ac.uk/~alexk/scscp.htm
19. Leroy, X.: Proving a compiler: Mechanized verification of program transformations and static analyses. Oregon Programming Languages Summer School (2010), http://cristal.inria.fr/~xleroy/courses/Eugene-2010/
20. OpenMath, http://www.openmath.org/
21. Saïbi, A.: Typing algorithm in type theory with inheritance. In: Proc. POPL 1997, pp. 292–301 (1997)
22. Symbolic Computation Software Composability Protocol, http://www.symbolic-computation.org/SCSCP
23. The Coq development team. The Coq proof assistant reference manual, http://coq.inria.fr/refman/
24. Wadler, P.: Views: A way for pattern matching to cohabit with data abstraction. In: POPL, pp. 307–313 (1987)

Computer Certified Efficient Exact Reals in Coq

Robbert Krebbers and Bas Spitters*

Radboud University Nijmegen

Abstract. Floating point operations are fast, but require continuous effort on the part of the user in order to ensure that the results are correct. This burden can be shifted away from the user by providing a library of *exact* analysis in which the computer handles the error estimates. We provide an implementation of the exact real numbers in the Coq proof assistant. This improves on the earlier Coq-implementation by O'Connor in two ways: we use dyadic rationals built from the machine integers and we optimize computation of power series by using approximate division. Moreover, we use type classes for clean mathematical interfaces. This appears to be the first time that type classes are used in heavy computation. We obtain over a 100 times speed up of the basic operations and indications for improving the Coq system.

1 Introduction

Real numbers cannot be represented exactly in a computer. Hence, in constructive analysis [1] one approximates real numbers by rational, or dyadic numbers. The real numbers are the completion of the rationals. This completion construction can be organized in a monad, a familiar construct from functional programming (Section 3). The completion monad provides an efficient combination of proving and computing [2]. In this way, O'Connor [3] implements exact real numbers and the transcendental functions on them in Coq.

A number of possible improvements in this implementation were already suggested in [4]. First, we can use Coq's new machine integers; see Section 2. Second, we can use dyadic rationals (that are numbers of the shape $n * 2^e$ for $n, e \in \mathbb{Z}$, also known as infinitary floats). Third, the implementation of power series can be improved by using approximate division. Here we carry out all three optimizations. Unfortunately, changing O'Connor's implementation to use the new machine integers was far from trivial, as he used a particular concrete representation of the rationals. To avoid this in the future, we provide an abstract specification of the dense set as *approximate rationals*; see Section 4.

In Section 4 we provide some abstract order theory culminating in the theory of approximate rationals. Section 5 deals with computing power series using dyadics. Section 6 describes Wolfram's algorithm to compute the square root of a real number. We finish with some benchmarks in Section 7.

* The research leading to these results has received funding from the European Union's 7th Framework Programme under grant agreement nr. 243847 (ForMath).

J.H. Davenport et al. (Eds.): Calculemus/MKM 2011, LNAI 6824, pp. 90–106, 2011.

2 The Coq-System

The Coq proof assistant is based on the calculus of inductive constructions [5,6], a dependent type theory with (co)inductive types; see [7,8]. In true Curry-Howard fashion, it is both a pure functional programming language with an expressive type system, and a language for mathematical statements and proofs. We highlight some aspects of Coq relevant for our development.

Types and propositions. Propositions in Coq are types [9,10], which themselves have types called *sorts*. Coq features a distinguished sort called Prop that one may choose to use as the sort for types representing propositions. The distinguishing feature of the Prop sort is that terms of non-Prop type may not depend on the values of inhabitants of Prop types (that is, proof terms). This regime of discrimination establishes a weak form of proof irrelevance, in that changing a proof can never affect the result of value computations. On a practical level, this lets Coq safely erase all Prop components when extracting certified programs to Ocaml or Haskell. We should note however, that in practice, Coq's extraction mechanism [11] is still very hard to use for programs with the complexity, in terms of depth of definitions, that we are interested in [12,13].

Equality, setoids, and rewriting. Because the Coq type theory lacks quotient types (as it would make type checking undecidable), one usually bases abstract structures on a *setoid* ('Bishop set'): a type equipped with an equivalence relation [1,14]. This leads to a naive set theory as described by Palmgren [15]. When the user attempts to substitute a given (sub)term using an equality, the system keeps track of, resolves, and combines proofs of equivalence [16].

The 'native' notion of equality in Coq, *Leibniz equality*, is that of terms being convertible, naturally reified as a proposition by the inductive type family eq with single constructor eq_refl : $\forall$ (T : Type)(x : T), x $\equiv$ x, where a $\equiv$ b is notation for eq T a b. Since convertibility is a congruence, a proof of a $\equiv$ b lets us substitute b for a anywhere inside a term without further conditions. Our interest is in more complicated equalities, so we diverge from Coq tradition and reserve = for setoid equality. Rewriting with = *does* give rise to side conditions. For instance, consider formal fractions of integers as a representation of rationals. Rewriting a subterm using such an equality is permitted only if the subterm is an argument of a function that has been proven to *respect* the equality. Such a function is called Proper, and that property must be proved for each function in whose arguments we wish to enable rewriting.

Type classes. Type classes have been a great success story in the Haskell functional programming language, as a means of organizing interfaces of abstract structures. Coq's type classes provide a superset of their functionality, but are implemented in a different way.

In Haskell and Isabelle, type classes and their instances are second class. They are handled as specialized syntactic constructs whose semantics are given specifically by the type class apparatus. By contrast, the expressivity of dependent types and inductive families as supported in Coq, combined with the use

of pre-existing technology in the system (namely proof search and implicit arguments) enable a *first class* type class implementation [17]: classes are ordinary record types ('dictionaries'), instances are ordinary constants of these record types (registered as *hints* with the proof search machinery), class constraints are ordinary implicit parameters, and instance resolution is achieved by augmenting the unification algorithm to invoke ordinary proof search for implicit arguments of class type. Thus, type classes in Coq are realized by relatively minor syntactic aids that bring together existing facilities of the theory and the system into a coherent idiom, rather than by introduction of a new category of qualitatively different definitions with their own dedicated semantics.

We use the algebraic hierarchy based on type classes and its abstract specification of $\mathbb{N}, \mathbb{Z}$ and $\mathbb{Q}$ described in [18]. Unfortunately, we should note that we have clearly met the efficiency problems connected to the current implementation of type classes in Coq. Luckily, these efficiency problems are limited to instance resolution which is only performed at compile time. Type classes have only a very minor effect on the computation time of type checked terms due to the absence of code inlining; see Section 7 for timings.

Virtual machine and machine integers. Coq includes a virtual machine [19], vm_compute, based on OCaml's virtual machine to allow efficient evaluation. Both the abstract machine and its compilation scheme have been proved correct, in Coq, with respect to the weak reduction semantics. However, we still need to extend our trusted core to a bigger kernel, as the *implementation* has not been formally verified.

Machine integers were also added to the Coq system [20]. The usual evaluation inside Coq (compute) uses a special inductive type for cyclic integers, but the virtual machine uses OCaml's machine integers. This allows for a big speed-up, for which we pay by having to trust (the virtual machine and) that OCaml treats these integers correctly. The time difference between computation with Coq's int and OCaml's Big_int is about a factor of 20 [21] on primality tests.

3 Metric Spaces

Having completed our brief description of the Coq-system, we now turn to O'Connor's formalization of exact real numbers [2]. Traditionally, a metric space is defined as a set X with a metric function $d : X \times X \to \mathbb{R}^+$ satisfying certain axioms. We use a more relaxed definition of a metric space that does not require the metric be a function; see also [22]. The metric is represented via a (respectful) ball relation $\mathbf{B} : \mathbb{Q}_+ \to X \to X \to \mathsf{Prop}$ satisfying:

$\mathsf{msp\_refl} : \forall x\, \varepsilon,\ \mathbf{B}_\varepsilon\ x\ x$
$\mathsf{msp\_sym} : \forall x\, y\, \varepsilon,\ \mathbf{B}_\varepsilon\ x\ y \to \mathbf{B}_\varepsilon\ y\ x$
$\mathsf{msp\_triangle} : \forall x\, y\, z\, \varepsilon_1\, \varepsilon_2,\ \mathbf{B}_{\varepsilon_1}\ x\ y \to \mathbf{B}_{\varepsilon_2}\ y\ z \to \mathbf{B}_{\varepsilon_1+\varepsilon_2}\ x\ z$
$\mathsf{msp\_closed} : \forall x\, y\, \varepsilon,\ (\forall \delta,\ \mathbf{B}_{\varepsilon+\delta}\ x\ y) \to \mathbf{B}_\varepsilon\ x\ y$
$\mathsf{msp\_eq} : \forall x\, y,\ (\forall \varepsilon,\ \mathbf{B}_\varepsilon\ x\ y) \to x = y$

The ball relation $\mathbf{B}_\varepsilon\ x\ y$ expresses that the points x and y are within ε of each other. We call this a ball relationship because the partially applied relation

$\mathbf{B}_\varepsilon^X\, x : X \to \mathsf{Prop}$ is a predicate that represents the closed ball of radius ε around the point x. For example, the ball relation on $\mathbb{Q}$ is $\mathbf{B}_\varepsilon^{\mathbb{Q}}\, x\, y := |x - y| \leq \varepsilon$.

We will introduce the completion of a metric space as a monad. In order to do this we will first introduce monads.

Monads. Moggi [23] recognized that many non-standard forms of computation may be modeled by monads[1]. Wadler [24] popularized their use in functional programming. Monads are now an established tool to structure computation with side-effects. For instance, programs with input X and output Y which have access to a mutable state S can be modeled as functions of type $X \times S \to Y \times S$, or equivalently $X \to (Y \times S)^S$. The type constructor $\mathfrak{M}Y := (Y \times S)^S$ is an example of a monad. Similarly, partial functions may be modeled by maps $X \to Y_\perp$, where $Y_\perp := Y + ()$ is a monad.

The formal definition of a (strong) monad is a triple $(\mathfrak{M}, \mathsf{return}, \mathsf{bind})$ consisting of a type constructor $\mathfrak{M}$ and two functions:

$$\begin{aligned} \mathsf{return} &: X \to \mathfrak{M}X \\ \mathsf{bind} &: (X \to \mathfrak{M}Y) \to \mathfrak{M}X \to \mathfrak{M}Y \end{aligned}$$

We will denote $\mathsf{return}\, x$ as $\hat{x}$, and $\mathsf{bind}\, f$ as $\check{f}$. These two operations must satisfy:

$$\begin{aligned} \mathsf{bind}\ \mathsf{return}\ a &= a \\ \check{f}\, \hat{a} &= f\, a \\ \check{f}\, (\check{g}\, a) &= \mathsf{bind}\ (\check{f} \circ g)\, a \end{aligned}$$

Completion monad. The completion of a metric space X is defined by:

$$\mathfrak{C}X := \{f : \mathbb{Q}_+ \to X \mid \forall \varepsilon_1\, \varepsilon_2,\ \mathbf{B}_{\varepsilon_1+\varepsilon_2}\, (f\, \varepsilon_1)\, (f\, \varepsilon_2)\}.$$

Given metric spaces X and Y, a function $f : X \to Y$ is *uniformly continuous* with *modulus* $\mu_f : Q_+ \to Q_+$ if:

$$\forall \varepsilon\, x_1\, x_2,\ \mathbf{B}_{\mu_f \varepsilon}\, x_1\ x_2 \to \mathbf{B}_\varepsilon\, (f\, x_1)\, (f\, x_2).$$

Completion is a monad on the category of metric spaces with uniformly continuous functions. The function $\mathsf{return} : X \to \mathfrak{C}X$ defined by $\lambda x\, \varepsilon,\ x$ is the embedding of a metric space in its completion. Moreover, a uniformly continuous function $f : X \to \mathfrak{C}Y$ with modulus μ_f can be lifted to operate on complete metric spaces as $\mathsf{bind}\, f : \mathfrak{C}X \to \mathfrak{C}Y$ defined by $\lambda x\, \varepsilon,\ f\, (x\, (\mu_f\, \frac{\varepsilon}{2}))\, \frac{\varepsilon}{2}$. In fact, the text above contains a white lie: we need a minor restriction to prelength spaces [3].

One advantage of this approach is that it helps us to work with simple representations. Let $\mathbb{R} := \mathfrak{C}\mathbb{Q}$. Then to specify a function from $\mathbb{R} \to \mathbb{R}$, we define a uniformly continuous function $f : \mathbb{Q} \to \mathbb{R}$, and obtain $\check{f} : \mathbb{R} \to \mathbb{R}$ as the required function. Hence, the completion monad allows us to do in a structured way what was already folklore in constructive mathematics: to work with simple, often decidable, approximations to continuous objects.

[1] In category theory one would speak about the Kleisli category of a (strong) monad.

4 Abstract Interfaces Using Type Classes

An important part of this work is to further develop the algebraic hierarchy based on type classes by Spitters and van der Weegen [18]. Especially, we have formalized some order theory and developed interfaces for mathematical operations common in programming languages such as shift and power. This layer of abstraction makes both proof engineering and programming more flexible: it avoids duplication of code, it introduces a canonical way to refer to operations and properties, both by names and notations, and it allows us to easily swap different implementations of number representations and their operations. First we will briefly recap the design decisions made in [18].

Algebraic structures are expressed in terms of a number of carrier sets, a number of relations and operations, and a number of laws that the operations satisfy. One way of describing such a structure is by a *bundled representation*: one uses a dependently typed record that contains the carrier, operations and laws. For example a semigroup can be represented as follows. (The fields sg_car and sg_proper support our explicit handling of naive set theory in type theory.)

```
Record SemiGroup : Type := {
  sg_car :> Setoid ;
  sg_op : sg_car → sg_car → sg_car ;
  sg_proper : Proper ((=) ⟹ (=) ⟹ (=)) sg_op ;
  sg_ass : ∀ x y z, sg_op x (sg_op y z) = sg_op (sg_op x y) z) }
```

However, this approach has some serious limitations, the most important one being a lack of support for *sharing* components. For example, suppose we group together two CommutativeMonoids in order to create a SemiRing. Now awkward hacks are necessary to establish equality between the carriers. A second problem is that if we stack up these records to represent higher structures the projection paths become increasingly long.

Historically these problems have been an acceptable trade-off because *unbundled representations*, in which the carrier and operations are parameterized, introduce even more problems.

```
Record SemiGroup {A} (e : A → A → Prop) (sg_op : A → A → A) : Prop := {
  sg_proper : Proper (e ⟹ e ⟹ e) sg_op ;
  sg_ass : ∀ x y z, e (sg_op x (sg_op y z)) (sg_op (sg_op x y) z) }
```

There is nothing to bind notation to, no structure inference, and declaring and passing requires too much manual bookkeeping. Spitters and van der Weegen have proposed a use of Coq's new type class machinery that resolves many of the problems of unbundled representations. Our current experiment confirms that this is a viable approach.

An alternative solution is provided by packed classes [25] which use an alternative, and older, implementation of a semblance of type classes: canonical structures. Yet another approach would use modules. However, as these are not

fist class, we would be unable to define, e.g. homomorphisms between algebraic structures.

An *operational type class* is defined for each operation and relation.

```
Class Equiv A := equiv: relation A.
Infix "=" := equiv: type_scope.
Class RingPlus A := ring_plus: A → A → A.
Infix "+" := ring_plus.
```

Now an algebraic structure is just a type class living in Prop that is parametrized by its carrier, relations and operations. This class contains all laws that the operations should satisfy. Since the operations are unbundled we can easily support sharing. For example let us consider the SemiRing interface.

```
Class SemiRing A {e : Equiv A} {plus: RingPlus A}
   {mult: RingMult A} {zero: RingZero A} {one: RingOne A} : Prop := {
  semiring_mult_monoid :> @CommutativeMonoid A e mult one ;
  semiring_plus_monoid :> @CommutativeMonoid A e plus zero ;
  semiring_distr :> Distribute (.*.) (+) ;
  semiring_left_absorb :> LeftAbsorb (.*.) 0 }.
```

Without type classes it would be a burden to manually carry around the carrier, relations and operations. However, because these parameters are just type class instances, the type class machinery will perform that job for us. For example,

```
Lemma example `{SemiRing R} x : 1 * x = x + 0.
```

The backtick instructs Coq to automatically insert implicit declarations, namely e plus mult zero one. It further lets us omit a name for the SemiRing R parameter itself as well. All of these parameters will be given automatically generated names that we will never refer to. Furthermore, instance resolution will automatically find instances of the operational type classes for the written notations. Thus the above is really:

```
Lemma example {R e plus mult zero one} {P : @SemiRing R e plus mult zero one} x :
  @equiv R e
    (@ring_mult R mult (@ring_one R one) x)
    (@ring_plus R plus x (@ring_zero R zero)).
```

The syntax :> in the definition of SemiRing declares certain fields as substructures. That means, a SemiRing can be seen as a CommutativeMonoid and each time a CommutativeMonoid instance is needed, a SemiRing can be used instead. This syntax should not be confused with the similar syntax for coercions in records (e.g. in the bundled representation of a SemiGroup on p. 94).

This approach to interfaces proved useful to formalize a standard algebraic hierarchy. Combined with category theory and universal algebra, $\mathbb{N}$ and $\mathbb{Z}$ are represented as interfaces specifying an initial SemiRing and initial Ring [18]. These abstract interfaces for the naturals and integers make it easier to change the

concrete representation in the future. No such simple specification for $\mathbb{Q}$ seems to exists, so we choose to specify it as the field of fractions of $\mathbb{Z}$. More precisely, $\mathbb{Q}$ is specified as a Field containing $\mathbb{Z}$ that moreover can be embedded into the field of fractions of $\mathbb{Z}$.

```
Inductive Frac R `{e : Equiv R} `{zero : RingZero R} : Type :=
  frac { num : R ; den : R ; den_nonzero : den ≠ 0 }.
Class RationalsToFrac (A : Type) := rationals_to_frac : ∀ B `{Integers B}, A → Frac B.
Class Rationals A {e plus mult zero one opp inv} `{U : !RationalsToFrac A} : Prop := {
  rationals_field :> @Field A e plus mult zero one opp inv ;
  rationals_frac :> ∀ `{Integers Z}, Injective (rationals_to_frac A Z) ;
  rationals_frac_mor :> ∀ `{Integers Z}, SemiRing_Morphism (rationals_to_frac A Z) ;
  rationals_embed_ints :> ∀ `{Integers Z}, Injective (integers_to_ring Z A) }.
```

4.1 Order Theory

To abstract from $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ and $\mathbb{R}$ and their various implementations, we provide a basic library for ordered algebraic structures. For example,

```
Class RingOrder `{Equiv A} `{RingPlus A} `{RingMult A} `{RingZero A}
    (o : Order A) := {
  ringorder_partialorder :> PartialOrder (≤) ;
  ringorder_plus :> `(OrderPreserving (z +));
  ringorder_mult : `(0 ≤ x → ∀ y, 0 ≤ y → 0 ≤ x * y) }.
```

To apply this to $\mathbb{N}$, which is merely a semiring, we introduce the, apparently new, notion of a SemiRingOrder. Every RingOrder is a SemiRingOrder.

```
Class SemiRingOrder `{Equiv A} `{RingPlus A} `{RingMult A} `{RingZero A}
    (o : Order A) := {
  srorder_partialorder :> PartialOrder (≤) ;
  srorder_plus : `(x ≤ y ↔ ∃ z, 0 ≤ z ∧ y = x + z) ;
  srorder_mult : `(0 ≤ x → ∀ y, 0 ≤ y → 0 ≤ x * y) }.
```

This allows us to refer by canonical names to lemmas as those shown below for $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ and the dyadics.

```
Lemma plus_compat x1 y1 x2 y2 : x1 ≤ y1 → x2 ≤ y2 → x1 + x2 ≤ y1 + y2 .
Lemma sprecedes_1_2 : 1 < 2.
```

For instances of $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ it is easy to define an order satisfying these interfaces:

```
Instance nat_precedes `{Naturals N} : Order N | 10 := λ x y, ∃ z, y = x + z.
```

However, often we encounter an a priori different order on a structure, most likely an order defined in Coq's standard library (like Nle on N). Therefore we prove that an arbitrary order satisfying these interfaces while also being a TotalOrder uniquely specifies the order on $\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{Q}$. For example:

```
Context `{Naturals N} `{Naturals N2} {f : N → N2} `{!SemiRing_Morphism f}
  {o1 : Order N} `{!SemiRingOrder o1} `{!TotalOrder o1}
  {o2 : Order N2} `{!SemiRingOrder o2} `{!TotalOrder o2}.
Global Instance: OrderEmbedding f.
```

Unfortunately Coq has no support to have an argument be 'inferred if possible, generalized otherwise'; see [18]. When declaring a parameter of RingOrder, one is often in a context where most of its components are already available. Usually, only the parameter Order has to be introduced. The current workaround in these cases involves providing names for components that are then never referred to, which is a bit awkward. In the above it would much nicer to write:

```
Context `{Naturals N} `{Naturals N2} {f : N → N2} `{!SemiRing_Morphism f}
  `{!SemiRingOrder N} `{!TotalOrder N} `{!SemiRingOrder N2} `{!TotalOrder N2}.
Global Instance: OrderEmbedding f.
```

4.2 Basic Operations

The operation nat_pow is most commonly, but inefficiently, defined as repeated multiplication and the operation shiftl is defined as repeated multiplication by 2. Instead we specify the desired behavior of these operations. This approach allows for different implementations for different number representations and avoids definitions and proofs becoming implementation dependent.

We introduce interfaces that specify the behavior of the operations abs, shiftl, nat_pow and int_pow. Again there are various ways of specifying these interfaces: with Σ-types, bundled or unbundled. In general, Σ-types are convenient for functions whose specification is easy, for example:

```
Class Abs A `{Equiv A} `{Order A} `{RingZero A} `{GroupInv A}
    := abs_sig: ∀ (x : A), { y : A | (0 ≤ x → y = x) ∧ (x ≤ 0 → y = -x)}.
Definition abs `{Abs A} := λ x : A, ` (abs_sig x).
```

However, for more complex operations, such as shiftl, such an interface is different from the usual mathematical specification because we cannot quantify over all possible input values. Now there are two ways: a bundled or an unbundled interface. Since these interfaces are not used for hierarchies the disadvantages of the latter do not apply. Let us first describe the former approach.

```
Class ShiftL A B `{Equiv A} `{Equiv B} `{RingOne A}
    `{RingPlus A} `{RingMult A} `{RingZero B} `{RingOne B} `{RingPlus B} := {
  shiftl : A → B → A ;
  shiftl_proper : Proper ((=) ⟹ (=) ⟹ (=)) shiftl ;
  shiftl_0 :> RightIdentity shiftl 0 ;
  shiftl_S : ∀ x n, shiftl x (1 + n) = 2 * shiftl x n }.
Infix "≪" := shiftl (at level 33, left associativity).
```

Although this interface seems reasonable, it does not work well in Coq. The simpl tactic which is used to simplify a goal will unfold occurrences of shiftl to their underlying definition (for example in case of BigN, the expression x ≪ n becomes

BigN.shiftl x n). This is rather inconvenient because Coq will then be unable to use lemmas concerning $\ll$ for rewriting. This problem is caused because shiftl is a projection of a record, which is in fact an ι-redex (reduction of pattern-matching over a constructed term) that will be unfolded by simpl. Currently there seems to be no way to adjust the behavior of simpl to remove this inconvenience. A similar problem was already observed in Ssreflect [26].

Instead we use an unbundled interface, which has a lot in common with our interfaces for algebraic structures. Now shiftl no longer contains an ι-redex.

```
Class ShiftL A B := shiftl: A → B → A.
Infix "≪" := shiftl (at level 33, left associativity).
Class ShiftLSpec A B (sl : ShiftL A B) `{Equiv A} `{Equiv B} `{RingOne A}
    `{RingPlus A} `{RingMult A} `{RingZero B} `{RingOne B} `{RingPlus B} := {
  shiftl_proper : Proper ((=) ⟹ (=) ⟹ (=)) (≪) ;
  shiftl_0 :> RightIdentity (≪) 0 ;
  shiftl_S : ∀ x n, x ≪ (1 + n) = 2 * x ≪ n }.
```

We do not specify shiftl as shiftl x n = x * 2 ^ n since on the dyadics we cannot take a negative power while we can shift by a negative integer.

4.3 Decision Procedures

The Decision type class collects types with a decidable equality [18].

```
Class Decision P := decide: sumbool P (¬ P).
```

Using this type class we can declare a parameter `{∀ x y, Decision (x ≤ y)} to describe a decider for $\leq$ and say decide (x ≤ y) to decide whether x ≤ y or not. This type class allows us to easily define additional deciders, like the one for the strict order. We have to be careful however. Consider the order on the dyadics.

```
Global Instance dy_precedes: Order Dyadic := λ (x y : Dyadic),
  ZtoQ (mant x) * 2 ^ (expo x) ≤ ZtoQ (mant y) * 2 ^ (expo y)
```

Now, decide (x ≤ y) is actually @decide Dyadic (x ≤ y) dyadic_dec, where dyadic_dec is the computational conclusion of the decision. Due to eager evaluation, and the absence of dead code removal, the second argument, x ≤ y, is also evaluated. Evaluation of this argument results in a conversion of x and y into Q, as described above. But since this argument is just a proposition it is later thrown away. We avoid this problem introducing a λ-abstraction.

```
Definition decide_rel `(R : relation A) {dec : ∀ x y, Decision (R x y)}
    (x y : A) : Decision (R x y) := dec x y.
```

We can now define:

```
Context `{!PartialOrder (≤) } {!TotalOrder (≤) } `{∀ x y, Decision (x ≤ y)}.
Global Program Instance sprecedes_dec: ∀ x y, Decision (x < y) | 9 := λ x y,
  match decide_rel (≤) y x with
  | left E ⇒ right _
  | right E ⇒ left _
  end.
```

4.4 Approximate Rationals

To make our implementation of the reals independent of the underlying dense set, we provide an abstract specification of *approximate rationals* inspired by the notion of *approximate fields* which is used in the Haskell implementation of the exact reals by Bauer and Kavler [27]. We provide an implementation of this interface by dyadics based on Coq's machine integers.

Our interface describes an ordered ring containing Z that is dense in Q. Here Z are the binary integers from Coq's standard library, and Q are the rationals based on these binary integers. We do not parametrize by arbitrary integer and rational implementations because they are hardly used for computation.

Also, for efficient computation, this interface contains the operations: approximate division, normalization, an embedding of Z, absolute value, power by N, shift by Z, and decision procedures for both equality and order.

```
Class AppDiv AQ := app_div : AQ → AQ → Z → AQ.
Class AppApprox AQ := app_approx : AQ → Z → AQ.
Class AppRationals AQ {e plus mult zero one inv} `{!Order AQ}
    {AQtoQ : Coerce AQ Q_as_MetricSpace} `{!AppInverse AQtoQ}
    {ZtoAQ : Coerce Z AQ} `{!AppDiv AQ} `{!AppApprox AQ}
  `{!Abs AQ} `{!Pow AQ N} `{!ShiftL AQ Z}
  `{∀ x y : AQ, Decision (x = y)} `{∀ x y : AQ, Decision (x ≤ y)} : Prop := {
 aq_ring :> @Ring AQ e plus mult zero one inv ;
 aq_order_embed :> OrderEmbedding AQtoQ ;
 aq_ring_morphism :> SemiRing_Morphism AQtoQ ;
 aq_dense_embedding :> DenseEmbedding AQtoQ ;
 aq_div : ∀ x y k, B_{2^k} ('app_div x y k) ('x / 'y) ;
 aq_approx : ∀ x k, B_{2^k} ('app_approx x k) ('x) ;
 aq_shift :> ShiftLSpec AQ Z (≪) ;
 aq_nat_pow :> NatPowSpec AQ N (^) ;
 aq_ints_mor :> SemiRing_Morphism ZtoAQ }.
```

O'Connor [2] keeps the size of the rational numbers small to avoid efficiency problems. He introduces a function approx x ϵ that yields the 'simplest' rational number between x $-$ ϵ and x $+$ ϵ. In our interface we modify the approx function slightly: app_approx x k yields an arbitrary element between x $- 2^k$ and x $+2^k$. Using this function we define the compress operation on the real numbers: compress := bind (λ ϵ, app_approx x (Qdlog2 ϵ)) such that compress x = x.

In Section 5 we will explain our choice of using a power of 2 to specify the precision of app_div and app_approx. In the remainder of this section we briefly describe our implementation by the dyadics.

The dyadic rationals are numbers of the shape $n * 2^e$ for $n, e \in \mathbb{Z}$. In order to remain independent of an integers implementation, we abstract over it. For our eventual implementation of the approximate rationals we use Coq's machine integers, bigZ. Now given an arbitrary integer implementation Int it is straightforward to define the dyadics. Here we will just show the ring operations.

```
Notation "x ↾ p" := (exist _ x p) (at level 20).
Record Dyadic := dyadic { mant : Int ; expo : Int }.
Infix "$" := dyadic (at level 80).
Global Instance dy_inject: Coerce Int Dyadic := λ x, x $ 0.
Global Instance dy_opp: GroupInv Dyadic := λ x, −mant x $ expo x.
Global Instance dy_mult: RingMult Dyadic := λ x y, mant x ∗ mant y $ expo x + expo y.
Global Instance dy_0: RingZero Dyadic := ('0:Dyadic).
Global Instance dy_1: RingOne Dyadic := ('1:Dyadic).
Global Program Instance dy_plus: RingPlus Dyadic := λ x y,
  if decide_rel (≤) (expo x) (expo y)
  then mant x + mant y ≪ (expo y − expo x) ↾ _ $ min (expo x) (expo y)
  else mant x ≪ (expo x − expo y) ↾ _ + mant y $ min (expo x) (expo y).
```

In this code shiftl has type Int → Int$^+$→ Int, where Int$^+$ is a Σ-type describing the non-negative elements of Int. Therefore, in the definition of dy_plus we have to equip expo y − expo x with a proof that it is in fact non-negative.

5 Power Series

Elementary transcendental functions as exp, sin, ln and arctan can be defined by their power series. If the coefficients of a power series are alternating, decreasing and have limit 0, then we obtain a fast converging sequence with an easy termination proof. For $-1 \leq x \leq 0$,

$$\exp\, x = \sum_{i=0}^{\infty} \frac{x^i}{i!}$$

is of this form. To approximate exp x with error ε we take the partial sum until $\frac{x^i}{i!} \leq \varepsilon$. In order to implement this efficiently we use a stream representing the series and define a function that sums the required number of elements. For example, the series 1, a, a^2, a^3, ... is defined by the following stream.

```
CoFixpoint powers_help (c : A) : Stream A := Cons c (powers_help (c ∗ a)).
Definition powers : Stream A := powers_help 1.
```

Streams in Coq, like lists in Haskell, are lazy. So, in the example the multiplications are accumulated.

Since Coq only allows structural recursion (and guarded co-recursion) it requires some work to convince Coq that our algorithm terminates. Intuitively, one would describe the limit as an upperbound of the required number of elements using the Exists predicate.

```
Inductive Exists A (P : Stream A → Prop) (x : Stream) : Prop :=
  | Here : P x → Exists P x
  | Further : Exists P (tl x) → Exists P x.
```

This approach leads to performance problems. The upperbound, encoded in unary format, may become very large while generally only a few terms are

necessary. Due to vm_compute's eager evaluation scheme, this unary number will be computed before summing the series. Instead we use LazyExists [28].

```
Inductive LazyExists A (P : Stream A → Prop) (x : Stream A) : Prop :=
  | LazyHere : P x → LazyExists P x
  | LazyFurther : (unit → LazyExists P (tl x)) → LazyExists P x.
```

O'Connor's InfiniteAlternatingSum s returns the real number represented by the infinite alternating sum over s, where the stream s is decreasing, non-negative and has limit 0. We have extended this in two ways. First, by generalizing some of the work to abstract structures. Second, as we do not have exact division on approximate rationals, we extended his algorithm to work with approximate division. The latter required changing InfiniteAlternatingSum s to InfiniteAlternatingSum n d which computes the infinite alternating sum of the stream $\lambda i, \frac{n_i}{d_i}$. This allows us to postpone divisions. Also, we have to determine both the length of the partial sum and the required precision of the divisions. To do so we find a k such that:

$$\mathbf{B}_{\frac{\varepsilon}{2}}\ (\mathsf{app\_div}\ n_k\ d_k\ (\mathsf{log}\frac{\varepsilon}{2k}) + \frac{\varepsilon}{2k})\ 0. \tag{1}$$

Now k is the length of the partial sum, and $\frac{\varepsilon}{2k}$ is the required precision of division. Using O'Connor's results we have verified that these values are correct and such a k indeed exists for a decreasing, non-negative stream with limit 0.

As noted in Section 4.4, we have specified the precision of division in powers of 2 instead of using a rational value. This allows us to replace (1) with:

$$\mathbf{B}_{\frac{\varepsilon}{2}}\ (\mathsf{app\_div}\ n_k\ d_k\ (\mathsf{log}\ \varepsilon - (k+1)) + 1 \ll (\mathsf{log}\ \varepsilon - (k+1)))\ 0.$$

Here k is the length of the partial sum, and 2^l, where $l = \mathsf{log}\ \varepsilon - (k+1)$, is the required precision of division. This variant can be implemented without any arithmetic on the rationals and is thus much more efficient.

This method gives us a fast way to compute the infinite alternating sum, in practice, only a few extra terms have to be computed and due to the approximate division the auxiliary results are kept as small as possible.

Using this method to compute infinite alternating sums we have so far implemented exp and arctan. Furthermore, we extend the exponential to its complete domain by repeatedly applying the following formula.

$$\mathsf{exp}\ x = (\mathsf{exp}(x \ll 1))^2 \tag{2}$$

Our tests have shown that reducing the input to a value between $-2^k \leq x \leq 0$ for $50 \leq k$ yields major performance improvements as the series will converge much faster. For higher precisions setting it to $75 \leq k$ gives even better results.

By defining arctan on $[0, 1)$, we can define the Machin-like formula

$$\pi := 176 * \mathsf{arctan}\frac{1}{57} + 28 * \mathsf{arctan}\frac{1}{239} - 48 * \mathsf{arctan}\frac{1}{682} + 96 * \mathsf{arctan}\frac{1}{12943}.$$

Since we do not have exact division on the approximate rationals, we see here the purpose of parameterizing infinite sums by two streams.

6 Square Root

We use Wolfram's algorithm [29, p.913] for computing the square root. Its complexity is linear, in fact it provides a new binary digit in each step. We aim to investigate Newton iteration in future work.

```
Context '(Pa : 1 ≤ a ≤ 4).
Fixpoint AQroot_loop (n : nat) : AQ * AQ :=
  match n with
  | O ⇒ (a, 0)
  | S n ⇒
    let (r, s) := AQroot_loop n in
    if decide_rel (≤) (s + 1) r
    then ((r − (s + 1)) ≪ (2:Z), (s + 2) ≪ (1:Z))
    else (r ≪ (2:Z), s ≪ (1:Z))
  end.
```

Three easy invariants allow us to prove this series converges to the square root.

```
Lemma AQroot_loop_invariant1 (n : nat) :
 snd (AQroot_loop n) * snd (AQroot_loop n) + 4 * fst (AQroot_loop n) = 4 * 4 ^ n * a.

Lemma AQroot_loop_invariant2 (n : nat) :
 fst (AQroot_loop n) ≤ 2 * snd (AQroot_loop n) + 4.
Lemma AQroot_loop_fst_bound (n : nat) :
 fst (AQroot_loop n) ≤ 2 ^ (3 + n).
```

Table 1. HASKELL, compiled with `ghc` version 6.12.1, using `-O2`

Expression	Decimals	O'Connor	Krebbers/Spitters
sin (sin (sin 1))	10,000	71s	5s
cos (10^{50})	10,000	2.7s	0.6s
tan $(\sqrt{2})$ + arctanh (sin 1)	500	133s	2.2s

Table 2. COQ trunk revision 13841

Expression	Decimals	O'Connor	Krebbers/Spitters
π	300	55s	0.8s
exp (exp (exp $(\frac{1}{2})$))	25	123s	0.23s
exp $\pi - \pi$	25	52s	0.1s
arctan π	25	134s	1.0s

7 Benchmarks

The first step in this research was to create a HASKELL prototype based on O'Connor's implementation of the real numbers in HASKELL [2]. The second step was to implement this prototype in COQ. Currently, our COQ development

contains the field operations, computation of power series, exp, arctan, π and the square root. Apart from the square root, the correctness of these operations has been verified in the Coq system.

In this section we present some benchmarks comparing the old and the new implementation, both in Haskell and Coq. All benchmarks have been carried out on an Intel Core Quad 2.4 GHz with 8GB of memory running Debian GNU/Linux with kernel 2.6.32. The sources of our developments can be found at `http://robbertkrebbers.nl/research/reals`.

Table 1 shows some benchmarks in Haskell with compiler optimizations enabled (`-O2`) and Table 2 compares our Coq implementation with O'Connor's. More extensive benchmarking shows that our Haskell implementation generally benefits from a 15 times speed up while the speed up in Coq is usually more than a 100 times. This difference is explained by the fact that O'Connor's Haskell implementation already used fast integers, while his Coq implementation did not. In the same times as shown in Table 2 for the old implementation, the new implementation is able to compute the first 2,000 decimals of π, 450 decimals of $\mathsf{exp}\ (\mathsf{exp}\ (\mathsf{exp}\ (\frac{1}{2})))$, 425 decimals of $\mathsf{exp}\ \pi - \pi$ and 85 decimals of $\mathsf{arctan}\ \pi$. This is an improvement of up to 18 times of the number of decimals.

It is interesting to notice that π and arctan benefit the least from our improvements, as we are unaware of an optimization similar to the squaring trick for exp (Section 5, Equation 2).

We conclude this section with a comparison between the performance of Wolfram's algorithm in Coq and Haskell. The Haskell prototype (without compiler optimizations) is quite fast, computing 10,000 iterations (giving 3,010 decimals) of $\sqrt{2}$ takes 0.2s. In Coq it takes 11.6s using type classes and 11.3s without type classes. Here we exclude the time spend on type class resolution. Thus type classes cause only a 3% performance penalty on computations.

Unfortunately, the Coq-implementation is slow compared to Haskell. Laurent Théry suggested that this is due to the representation of the fast integers, which uses a tree with a fixed depth and when the size of the integer becomes too big uses a less optimal representation. Increasing the size of the tree representation and avoiding an inefficiency in the implementation of shifts reduces this time to 7.5s.

8 Conclusions and Related Work

We have greatly improved the performance of real number computation in Coq using Coq's new machine integers. We produced highly structured and abstract code using type classes with no apparent performance penalty. Moreover, Coq's notation mechanism combined with unicode characters gives nicely readable statements and proofs. Type classes were a great help in our work. However, the current implementation of instance resolution is still experimental and at times too slow (at compile time). Canonical structures provide an alternative, and partially complementary, implementation of type classes [30]. By choice, canonical structures restrict to deterministic proof search, this makes them more efficient,

but also somewhat more intricate to use. The use of canonical structures by the SSREFLECT team [25] makes it plausible that with some effort we could have used canonical structures for our work instead. However, the SSREFLECT-library is currently not suited for setoids which are crucial to us. The integration of unification hints [31] into COQ may allow a tighter integration of type classes and canonical structures.

We needed to adapt our correctness proofs to prevent the virtual machine from eagerly evaluating them. Lazy evaluation for Prop would have allowed us to use the original proofs.

The experimental native_compute performs evaluation by compilation to native OCAML code. This approach uses the OCAML compiler available and is interesting for heavy compilation. Our first experiments indicate a 10 times speed up with Wolfram iteration. Unfortunately, native_compute does not work with COQ trunk yet, so we were unable to test it with our implementation of the reals.

The FLOCQ project [32] formalizes floating-points in COQ. It provides a library of theorems on a multi-radix multi-precision arithmetic and supports efficient numerical computations inside COQ. However, the current library is still too limited for our purposes, but in the future it should be possible to show that they form an instance of our approximate rationals. This may allow us to gain some speed by taking advantage of fine grained algorithms on the floats instead of our more straightforward ones.

The encoding of real numbers as streams of 'bits' is potentially interesting. However, currently there is a big difference in performance. The computation of 37 decimals of the square root of 1/2 by Newton iteration [33], using the framework described in [34,35], took 12s. This should be compared with our use of the Wolfram iteration, which gives only linear convergence, but with which we nevertheless obtain 3,000 decimals in in a similar time. On the other hand, the efficiency of π in their framework is comparable with ours. Berger [36], too, uses co-induction for exact real computation.

The present work is part of a larger program to use constructive mathematics based on type theory as a programming language for exact analysis. This should culminate in a numerical ODE-solver.

Acknowledgements We thank Eelis van der Weegen for many discussions and Pierre Letouzey and Matthieu Sozeau for closing some of our bug reports. We are grateful to the anonymous referees who provided some helpful suggestions.

References

1. Bishop, E.A.: Foundations of constructive analysis. McGraw-Hill, New York (1967)
2. O'Connor, R.: A Monadic, Functional Implementation of Real Numbers. MSCS 17(1), 129–159 (2007)
3. O'Connor, R.: Certified Exact Transcendental Real Number Computation in Coq. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 246–261. Springer, Heidelberg (2008)

4. O'Connor, R., Spitters, B.: A computer verified, monadic, functional implementation of the integral. TCS 411(37), 3386–3402 (2010)
5. Coquand, T., Huet, G.: The Calculus of Constructions. Information and Computation 76(2-3), 95–120 (1988)
6. Coquand, T., Paulin, C.: Inductively defined types. In: Martin-Löf, P., Mints, G. (eds.) COLOG 1988. LNCS, vol. 417, pp. 50–66. Springer, Heidelberg (1990)
7. Coq Development Team: The Coq Proof Assistant Reference Manual. INRIA-Rocquencourt (2008)
8. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Texts in TCS. Springer, Heidelberg (2004)
9. Martin-Löf, P.: An intuitionistic theory of types. In: Twenty-five years of constructive type theory. Oxford Logic Guides, vol. 36, pp. 127–172. OUP (1998)
10. Martin-Löf, P.: Constructive Mathematics and Computer Science. In: Logic, Methodology and the Philosophy of Science VI. Studies in Logic and the Foundations of Mathematics, vol. 104, pp. 153–175 (1982)
11. Letouzey, P.: Extraction in Coq: An Overview. In: Beckmann, A., Dimitracopoulos, C., Löwe, B. (eds.) CiE 2008. LNCS, vol. 5028, pp. 359–369. Springer, Heidelberg (2008)
12. Cruz-Filipe, L., Spitters, B.: Program Extraction from Large Proof Developments. In: Basin, D., Wolff, B. (eds.) TPHOLs 2003. LNCS, vol. 2758, pp. 205–220. Springer, Heidelberg (2003)
13. Cruz-Filipe, L., Letouzey, P.: A Large-Scale Experiment in Executing Extracted Programs. Electronic Notes in Theoretical Computer Science 151(1), 75–91 (2006)
14. Hofmann, M.: Extensional constructs in intensional type theory. CPHC/BCS Distinguished Dissertations. Springer, Heidelberg (1997)
15. Palmgren, E.: Constructivist and Structuralist Foundations: Bishops and Lawveres Theories of Sets. Technical Report 4, Mittag-Leffler (2009)
16. Sozeau, M.: A New Look at Generalized Rewriting in Type Theory. Journal of Formalized Reasoning 2(1), 41–62 (2009)
17. Sozeau, M., Oury, N.: First-class type classes. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 278–293. Springer, Heidelberg (2008)
18. Spitters, B., van der Weegen, E.: Type classes for mathematics in type theory. MSCS, special issue on Interactive theorem proving and the formalization of mathematics (2011)
19. Grégoire, B., Leroy, X.: A compiled implementation of strong reduction. In: ICFP, pp. 235–246 (2002)
20. Armand, M., Grégoire, B., Spiwack, A., Théry, L.: Extending Coq with imperative features and its application to SAT verification. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 83–98. Springer, Heidelberg (2010)
21. Spiwack, A.: Verified Computing in Homological Algebra, A Journey Exploring the Power and Limits of Dependent Type Theory. PhD thesis, INRIA (2011)
22. Richman, F.: Real numbers and other completions. Mathematical Logic Quarterly 54(1), 98–108 (2008)
23. Moggi, E.: Computational lambda-calculus and monads. In: LICS, pp. 14–23 (1989)
24. Wadler, P.: Monads for functional programming. In: Proceedings of the Marktoberdorf Summer School on Program Design Calculi (August 1992)

25. Garillot, F., Gonthier, G., Mahboubi, A., Rideau, L.: Packaging mathematical structures. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 327–342. Springer, Heidelberg (2009)
26. Gonthier, G., Mahboubi, A., Tassi, E.: A Small Scale Reflection Extension for the Coq system. Technical Report RR-6455, INRIA (2008)
27. Bauer, A., Kavkler, I.: A constructive theory of continuous domains suitable for implementation. Annals of Pure and Applied Logic 159(3), 251–267 (2009)
28. O'Connor, R.: Incompleteness and Completeness: Formalizing Logic and Analysis in Type Theory. PhD thesis, Radboud University Nijmegen (2009)
29. Wolfram, S.: A new kind of science. Wolfram Media (2002)
30. Gonthier, G., Ziliani, B., Nanevski, A., Dreyer, D.: Making ad hoc proof automation less ad hoc (2011)
31. Asperti, A., Ricciotti, W., Coen, C., Tassi, E.: Hints in Unification. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 84–98. Springer, Heidelberg (2009)
32. Boldo, S., Melquiond, G.: Flocq: A unified library for proving floating-point algorithms in Coq. In: Proc 20th IEEE Symposium on Computer Arithmetic (2011)
33. Julien, N., Pasca, I.: Formal Verification of Exact Computations Using Newton's Method. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 408–423. Springer, Heidelberg (2009)
34. Bertot, Y.: Affine functions and series with co-inductive real numbers. MSCS 17(1), 37–63 (2007)
35. Julien, N.: Certified Exact Real Arithmetic Using Co-induction in Arbitrary Integer Base. In: Garrigue, J., Hermenegildo, M.V. (eds.) FLOPS 2008. LNCS, vol. 4989, pp. 48–63. Springer, Heidelberg (2008)
36. Berger, U.: From coinductive proofs to exact real arithmetic. In: Grädel, E., Kahle, R. (eds.) CSL 2009. LNCS, vol. 5771, pp. 132–146. Springer, Heidelberg (2009)

A Foundational View on Integration Problems

Florian Rabe[1], Michael Kohlhase[1], and Claudio Sacerdoti Coen[2]

[1] Computer Science, Jacobs University, Bremen (DE)
initial.lastname@jacobs-university.de
[2] Department of Computer Science, University of Bologna (IT)
sacerdot@cs.unibo.it

Abstract. The integration of reasoning and computation services across system and language boundaries is a challenging problem of computer science. In this paper, we use integration for the scenario where we have two systems that we integrate by moving problems and solutions between them. While this scenario is often approached from an engineering perspective, we take a foundational view. Based on the generic declarative language MMT, we develop a theoretical framework for system integration using theories and partial theory morphisms. Because MMT permits representations of the meta-logical foundations themselves, this includes integration across logics. We discuss safe and unsafe integration schemes and devise a general form of safe integration.

1 Introduction

The aim of integrating Computer Algebra Systems (CAS) and Deduction Systems (DS) is twofold: to bring the efficiency of CAS algorithms to DS (without sacrificing correctness) and to bring the correctness assurance of the proof theoretic foundations of DS to CAS computations (without sacrificing efficiency). In general, the integration of computation and reasoning systems can be organized either by extending the internals of one system by methods (data structure and algorithms) from the other, or by passing representations of mathematical objects and system state between independent systems, thus delegating parts of the computation to more efficient or secure platforms. We will deal with the latter approach here, which again has two distinct sets of problems. The first addresses engineering problems and revolves about communication protocol questions like shared state, distributed garbage collection, and translating input syntaxes of the different systems. The syntax questions have been studied extensively in the last decade and led to universal content markup languages languages for mathematics like MathML and OpenMath to organize communication. The second set of problems comes from the fact that passing mathematical objects between systems can only be successful if their meaning is preserved in the communication. This meaning is given via logical consequence in the logical system together with the axioms and definitions of (or inscribed in) the respective systems.

We will address this in the current paper, starting from the observation that content level communication between mathematical systems, to be effective, cannot always respect logical consequence. On the other hand, there is the problem

J.H. Davenport et al. (Eds.): Calculemus/MKM 2011, LNAI 6824, pp. 107–122, 2011.

of trusting the communication itself, that boils down to studying the preservation of logical consequence. Surprisingly, this problem has not received in the literature the attention it deserves. Moreover, the problem of faithful safe communication, which preserves not only the consequence relation but also the intuitive meaning of a formal object, is not even always perceived as a structural problem of content level languages.

For example, people with a strong background in first order logic tend to assume that faithful and safe communication can always be achieved simply by strengthening the specifications; others believe that encoding logical theories is already sufficient for safe communication and do not appreciate that the main problem is just moved to faithfulness. Several people from the interactive theorem proving world have raised concerns about trusting CAS and solved the issue by re-checking the results or the traces of the computation (here called proof sketches). Sometimes this happens under the assumption that the computation is already correct and just needs to be re-checked, neglecting the interesting case when the proof sketch cannot be refined to a valid proof (or computation) without major patching (see [Del99] for a special case).

In this paper, we first give a categorization of integration problems and solutions. Then we derive an integration framework by adding some key innovations to the MMT language, a Module system for Mathematical Theories described in [RK11]. MMT can be seen as a generalization of OpenMath and as a formalized core of OMDoc. Of course, any specific integration task requires a substantial amount of work — irrespective of the framework used. But our framework guides and structures this effort, and can implement all the generic aspects. In fact, current integration tasks typically involve setting up an ad-hoc framework for exactly that reason.

We sketch the MMT framework first in Sect. 2. In Sect. 3, we analyze the integration problem for mathematical systems from a formal position. Then we describe how integration can be realized our framework using partial MMT theory morphisms in Sect. 4. Finally, Sect. 5 discusses related work and Sect. 6 concludes the paper.

2 The MMT Language

Agreeing on a common syntax like OpenMath is the first step towards system integration. This already enables a number of structural services such as storage and transport or editing and browsing that they do not depend on the semantics of the processed expressions. But while we have a good solution for a joint syntax, it is significantly harder to agree on a joint semantics. Fixing a semantics for a system requires a foundational commitment that excludes systems based on other foundations. The weakness of the (standard) OpenMath content dictionaries can be in part explained by this problem: The only agreeable content dictionaries are those where any axioms (formal or informal) are avoided that would exclude some foundations.

MMT was designed to overcome this problem by placing it in between frameworks like OpenMath and OMDoc on the one hand and logical frameworks

like LF and CIC on the other hand. The basic idea is that a system's foundation itself is represented as a content dictionary. Thus, both meta and object language are represented uniformly as MMT *theories*. Furthermore, *theory morphisms* are employed to translate between theories, which makes MMT expressive enough to represent translation between meta-languages and thus to support cross-foundation integration. As MMT permits the representation of logics as theories and internalizes the meta-relation between theories, this provides the starting point to analyze the cross-foundation integration challenge within a formal framework.

Syntax. We will work with a very simple fragment of the MMT language that suffices for our purposes, and refer to [RK11] for the full account. It is given by the following grammar where $[-]$ denotes optional parts and T, v, c, and x are identifiers:

Theory graph	γ	::=	$\cdot \mid \gamma,\ T \stackrel{[T]}{=} \{\vartheta\} \mid \gamma,\ v : T \to T \stackrel{v}{=} \{\sigma\}$
Theory body	ϑ	::=	$\cdot \mid \vartheta,\ c\,[: O]\,[= O']$
Morphism body	σ	::=	$\cdot \mid \sigma,\ c \mapsto O$
Objects	O	::=	OpenMath objects
Morphisms	μ	::=	$v \mid id_T \mid \mu \circ \mu$
Contexts	C	::=	$x_1 : O_1, \ldots, x_n : O_n$
Substitutions	s	::=	$x_1 := O_1, \ldots, x_n := O_n$

In particular, we omit the module system of MMT that permits imports between theories.

$T \stackrel{L}{=} \{\vartheta\}$ declares a *theory* T with *meta-theory* L defined by the list ϑ of symbol declarations. The intuition of meta-theories is that L is the meta-language that declares the foundational symbols used to type and define the symbol declarations in ϑ.

All *symbol declarations* in a theory body are of the form $c : O = O'$. This declares a new symbol c where both the type O and the definiens O' are optional. If given, they must be T-objects, which are defined as follows. A symbol is called *accessible* to T if it is declared in T or accessible to the meta-theory of T. An OpenMath object is called a *T-object* if it only uses symbols that are accessible to T.

Example 1. Consider the natural numbers defined within the calculus of constructions (see [BC04]). We represent this in MMT using a theory `CIC` declaring untyped, undefined symbols such as `Type`, λ and $\to$. Then `Nat` is defined as a theory with meta-theory `CIC` giving symbol declarations such as `N` : `OMS`(`cd` = `CIC`, `name` = `Type`) or `succ` : `OMA`(`OMS`(`cd` = `CIC`, `name` = $\to$), `OMS`(`cd` = `Nat`, `name` = `N`), `OMS`(`cd` = `Nat`, `name` = `N`)).

S-contexts C are lists of variable declarations $\ldots, x_i : O_i, \ldots$ for S-objects O_i. *S-substitutions* s for an S-context C are lists of variable assignments $\ldots, x_i := o_i, \ldots$. In an object O in context C, exactly the variables in C may occur freely; then for a substitution s for C, we write $O[s]$ for the result of replacing every free occurrence of x_i with o_i.

Relations between MMT theories are expressed using theory morphisms. Given two theories S and T, a *theory morphism* from S to T is declared using $v : S \to T \stackrel{l}{=} \{\sigma\}$. Here σ must contain one assignment $c \mapsto O$ for every symbol c declared in the body of S, and for some T-objects O. If S and T have meta-theories L and M, then v must also include a meta-morphism $l : L \to M$.

Every $v : S \to T \stackrel{l}{=} \{\sigma\}$ induces a *homomorphic extension* $v(-)$ that maps S-objects to T-objects. $v(-)$ is defined by induction on the structure of OpenMath objects. The base case $v(c)$ for a symbol c is defined as follows: If c is accessible to the meta-theory of S, we put $v(c) := l(c)$; otherwise, we must have $c \mapsto O$ in σ, and we put $v(c) := O$. $v(-)$ also extends to contexts and substitutions in the obvious way.

By experimental evidence, all declarative languages for mathematics currently known can be represented faithfully in MMT. In particular, MMT uses the Curry-Howard representation [CF58, How80] of propositions as types and proofs as terms. Thus, an axiom named a asserting F is a special cases of a symbol a of type F, and a theorem named t asserting F with proof p is a special case of a symbol t with type F and definiens p. All inference rules needed to form p, are symbols declared in the meta-theory.

Semantics. The use of meta-theories makes the logical foundation of a system part of an MMT theory and makes the syntax of MMT foundation-independent. The analogue for the semantics is more difficult to achieve: The central idea is that the semantics of MMT is parametric in the semantics of the foundation.

To make this precise, we call a theory without a meta-theory *foundational*. A *foundation* for MMT consists of a foundational theory L and two judgments for typing and equality of objects:

- $\gamma; C \vdash_T O : O'$ states that O is a T-object over C typed by the T-object O',
- $\gamma; C \vdash_T O = O'$ states the equality of two T-objects over C,

defined for an arbitrary theory T declared in γ with meta-theory L. In particular, MMT does not distinguish terms, types, and values at higher universes — all expressions are OpenMath objects with an arbitrary binary typing relation between them. We will omit C when it is empty.

These judgments are similar to those used in almost all declarative languages, except that we do not commit to a particular inference system — all rules are provided by the foundation and are transparent to MMT except for the rules for the base cases of T-objects:

$$\frac{T \stackrel{L}{=} \{\vartheta\} \text{ in } \gamma \quad c : O = O' \text{ in } \vartheta}{\gamma \vdash_T c : O}\,\mathcal{T}_: \qquad \frac{T \stackrel{L}{=} \{\vartheta\} \text{ in } \gamma \quad c : O = O' \text{ in } \vartheta}{\gamma \vdash_T c = O'}\,\mathcal{T}_=$$

and accordingly if O or O' are omitted. For example, adding the usual rules for the calculus of constructions yields a foundation for the foundational theory `CIC`.

Given a foundation, MMT defines (among others) the judgments

- $\gamma \vdash \mu : S \to T$ states that μ is a theory morphism from S to T,
- if $\gamma \vdash \mu_i : S \to T$, then $\gamma \vdash \mu_1 = \mu_2$ states that $\vdash_T \mu_1(c) = \mu_2(c)$ for all symbols c that are accessible to S,

- $\gamma \vdash_S s : C$ states that s is a well-typed for C, i.e., for every $x_i := o_i$ in s and $x_i : O_i$ in C, we have $\gamma \vdash_S o_i : O_i$,
- $\gamma \vdash \mathcal{G}$ states that $\mathcal{G}$ is a well-formed theory graph.

In the sequel, we will omit γ if it is clear from the context.

The most important MMT rule for our purposes is the rule that permits adding an assignment to a theory morphism: If S contains a declaration $c : O_1 = O_2$, then a theory morphism $v : S \to T \stackrel{l}{=} \{\sigma\}$ may contain an assignment $c \mapsto O$ only if $\vdash_T O : v(O_1)$ and $\vdash_T O = v(O_2)$. The according rule applies if c has no type or no definiens. Of course, this means that assignments $c \mapsto O$ are redundant if c has a definiens; but it is helpful to state the rule in this way to prepare for our definitions below.

Due to these rules, we obtain that if $\gamma \vdash \mu : S \to T$ and $\vdash_S O : O'$ or $\vdash_S O = O'$, then $\vdash_T \mu(O) : \mu(O')$ and $\vdash_T \mu(O) = \mu(O')$, respectively. Thus, typing and equality are preserved along theory morphisms.

Due to the Curry-Howard representation, this includes the preservation of provability: $\vdash_T p : F$ states that p is a well-formed proof of F in T. And if S contains an axiom $a : F$, a morphism μ from S to T must map a to a T-object of type $\mu(F)$, i.e., to a T-proof of $\mu(F)$. This yields the well-known intuition of a theory morphism. In particular, if μ is the identity on those symbols that do not represent axioms, then $\vdash \mu : S \to T$ implies that every S-theorem is an T-theorem.

MMT is parametric in the particular choice of type system — any type system can be used by giving the respective meta-theory. The type systems may themselves by defined in a further meta-theory. For example, many of our actual encodings are done with the logical framework LF [HHP93] as the ultimate meta-theory. The flexibility to use MMT with or without a logical framework that takes care of all typing aspects is a particular strength of MMT.

3 Integration Challenges

In this section, we will develop some general intuitions about system integration and then give precise definitions in MMT. A particular strength of MMT is that we can give these precise definitions without committing to a particular foundational system and thus without loss of generality.

The typical integration situation is that we have two systems $\mathcal{S}_i$ for $i = 1, 2$ that implement a shared specification *Spec*. For example, these systems can be computer algebra systems or (semi-)automated theorem provers. Our integration goal is to move problems and results between $\mathcal{S}_1$ and $\mathcal{S}_2$.

Specifications and Systems. Let us first assume a single system $\mathcal{S}$ implementing *Spec*, whose properties are given by logical consequence relations $\Vdash_{Spec}$ and $\Vdash_{\mathcal{S}}$. We call $\mathcal{S}$ *sound* if $\Vdash_{\mathcal{S}} F$ implies $\Vdash_{Spec} F$ for every formula F in the language of *Spec*. Conversely, we call $\mathcal{S}$ *complete* if $\Vdash_{Spec} F$ implies $\Vdash_{\mathcal{S}} F$.

While these requirements seem quite natural at first, they are too strict for practical purposes. It is well-known that soundness fails for many CASs, which

compute wrong results by not checking side conditions during simplification. Reasons for incompleteness can be theoretical — e.g., when $\mathcal{S}$ is a first-order prover and *Spec* a higher-order specification — or practical — e.g., due to resource limitations.

Moreover, soundness also fails in the case of underspecification: $\mathcal{S}$ is usually much stronger than *Spec* because it must commit to concrete definitions and implementations for operations that are loosely specified in *Spec*. A typical example is the representation of undefined terms (see [Far04] for a survey of techniques). If *Spec* specifies the rational numbers using in particular $\forall x.x \neq 0 \Rightarrow x/x = 1$, and $\mathcal{S}$ defines $1/0 = 2/0 = 0$, then $\mathcal{S}$ is not sound because $1/0 = 2/0$ is not a theorem of *Spec*.

We can define the above notions in MMT as follows. A *specification Spec* is an MMT theory; its meta-theory (if any) is called the *specification language*. A system implementing *Spec* consists of an MMT theory $\mathcal{S}$ and an MMT theory morphism $v : Spec \to \mathcal{S}$; the meta-theory of $\mathcal{S}$ (if any) is called the *implementation language*. With this definition and using the Curry-Howard representation of MMT, we can provide a deductive system for the consequence relations used above: $\Vdash_{Spec} F$ iff there is a p such that $\vdash_{Spec} p : F$; and accordingly for $\Vdash_{\mathcal{S}}$.

In the simplest case, the morphism v is an inclusion, i.e., for every symbol in *Spec*, $\mathcal{S}$ contains a symbol of the same name. Using an arbitrary morphism v provides more flexibility, for example, the theory of the natural numbers with addition and multiplication implements the specification of monoids in two different ways via two different morphisms.

Example 2. We use a theory for second-order logic as the specification language; it declares symbols for $\forall$, $=$, etc. $Spec = Nat$ is a theory for the natural numbers; it declares symbols N, 0 and *succ* as well as one symbol $a : F$ for each Peano axiom F.

For the implementation language, we use a theory **ZF** for ZF set theory; it has meta-theory first-order logic and declares symbols for **set**, $\in$, $\varnothing$, etc. Then we can implement the natural numbers in a theory $\mathcal{S} = \mathbf{Nat}$ declaring, e.g., a symbol **0** defined as $\varnothing$, a symbol **succ** defined such that $\mathbf{succ}(n) = n \cup \{n\}$, and prove one theorem $a : F = p$ in $\mathcal{S}$ for each Peano axiom. Note that **Nat** yields theorems about the natural numbers that cannot be expressed in *Spec*, for example $\Vdash_{\mathbf{ZF}} 0 \in 1$. We obtain a morphism $\mu_1 : Nat \to \mathbf{Nat}$ using $N \mapsto \mathbf{N}$, $0 \mapsto \mathbf{0}$ etc.

Continuing Ex. 1, we obtain a different implementation $\mu_2 : Nat \to \texttt{Nat}$ using $N \mapsto \texttt{N}$, $0 \mapsto \texttt{0}$ etc.

To capture practice in formal mathematics, we have to distinguish between the definitional and the axiomatic method. The *axiomatic* method fixes a formal system L and then describes mathematical notions in L-theories T using free symbols and axioms. T is interpreted in models, which may or may not exist. This is common in model theoretical logics, especially first-order logic, and in algebraic specification. In MMT, T is represented as a theory with meta-theory L and with only undefined constants. In Ex. 2, L is second-order logic and T is *Spec*.

The *definitional* method, on the other hand, fixes a formal system L together with a minimal theory T_0 and then describes mathematical notions using definitional extensions T of T_0. The properties of the notions defined in T_0 are derived as theorems. The interpretation of T is uniquely determined given a model of T_0. This is common in proof theoretical logics, especially LCF-style proof assistants, and in set theory. In Ex. 2, L is first-order logic, T_0 is ZF, and T is $\mathcal{S}$.

Types of Integration. Let us now consider a specification *Spec* and two implementations $\mu_i : Spec \to \mathcal{S}_i$. To simplify the notation, we will write $\vdash$ and $\vdash_i$ instead of $\vdash_{Spec}$ and $\vdash_{\mathcal{S}_i}$. We first describe different ways how to integrate $\mathcal{S}_1$ and $\mathcal{S}_2$ intuitively.

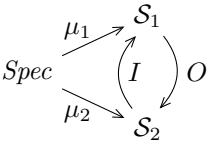
Borrowing means to use $\mathcal{S}_1$ to prove theorems in the language of $\mathcal{S}_2$. Thus, the input to $\mathcal{S}_1$ is a conjecture F and the output is an expression $\vdash_1 p : F$. In general, since MMT does not prescribe a calculus for proofs, the object p can be a formal proof term, a certificate, proof sketch, or simply a yes/no answer.

Computation means to reuse a $\mathcal{S}_1$ computation in $\mathcal{S}_2$. Thus, the input of $\mathcal{S}_1$ is an expression t, and the output is a proof p with an expression t' such that $\vdash_1 p : t = t'$. To be useful, t' should be simpler than t in some way, e.g., maximally simplified or even normalized.

Querying means answering a query in $\mathcal{S}_1$ and transferring the results to $\mathcal{S}_2$. This is similar to borrowing in that the input to $\mathcal{S}_1$ is a formula F. However, now F may contain free variables, and the output is not only a proof p but also a substitution s for the free variables such that $\vdash_1 p : F[s]$.

In all cases, a translation I must be employed to translate the input from $\mathcal{S}_1$ to $\mathcal{S}_2$. Similarly, we need a translation O in the opposite direction to translate the output t' and s and (if available) p from $\mathcal{S}_2$ to $\mathcal{S}_1$.

To define these integration types formally in MMT, we first note that borrowing is a special case of querying if F has no free variables. Similarly, computation is a special case of querying if F has the form $t = X$ for a variable X that does not occur in t.



To define querying in MMT, we assume a specification, two implementations, and morphisms I and O as on the right. I and O must satisfy $O \circ I = id_{\mathcal{S}_2}$, $O \circ \mu_1 = \mu_2$, and $I \circ \mu_2 = \mu_1$. Then we obtain the following *general form of an integration problem*: Given an $\mathcal{S}_2$-context C and a query $C \vdash_2 ? : F$ (where ? denotes the requested proof), find a substitution $\vdash_1 s : I(C)$ and a proof $\vdash_1 p : I(F)[s]$. Then MMT guarantees that $\vdash_2 O(p) : F[O(s)]$ so that we obtain $O(s)$ as the solution. Moreover, only the existence of O is necessary but not O itself — once a proof p is found in $\mathcal{S}_1$, the existence of O ensures that F is true in $\mathcal{S}_2$, and it is not necessary to translate p to $\mathcal{S}_2$.

We call the above scenario *safe bidirectional communication* between $\mathcal{S}_1$ and $\mathcal{S}_2$ because I and O are theory morphisms and thus guarantee that consequence and truth are preserved in both directions. This scenario is often implicitly assumed by people coming from the first-order logic community. Indeed, if $\mathcal{S}_1$ and $\mathcal{S}_2$ are automatic or interactive theorem provers for first-order logic, then the logic of the two systems is the same and both $\mathcal{S}_1$ and $\mathcal{S}_2$ are equal to *Spec*.

If we are only interested in *safe directed communication*, i.e., transferring results from $\mathcal{S}_1$ to $\mathcal{S}_2$, then it is sufficient to require only O. Indeed, often μ_2 is an inclusion, and the input parameters C and F, which are technically $\mathcal{S}_2$-objects, only use symbols from *Spec*. Thus, they can be moved directly to *Spec* and $\mathcal{S}_1$, and I is not needed.

Similarly, the substitution s can often be stated in terms of *Spec*. In that case, O is only needed to translate the proof p. If the proof translation is not feasible, O may be omitted as well. Then we speak of *unsafe communication* because we do not have a guarantee that the communication of results is correct. For example, let $\mathcal{S}_1$ and $\mathcal{S}_2$ be two CASs, that may compute wrong results by not checking side conditions during simplification. Giving a theory morphism O means that the "bugs" of the system $\mathcal{S}_1$ must be "compatible" with the "bugs" of $\mathcal{S}_2$, which is quite unlikely.

The above framework for safe communication via theory morphisms is particularly appropriate for the integration of axiomatic systems. However, if $\mathcal{S}_1$ and $\mathcal{S}_2$ employ different mathematical foundations or different variants of the same foundation, it can be difficult to establish the necessary theory morphisms. In MMT, this means that $\mathcal{S}_1$ and $\mathcal{S}_2$ have different meta-theories so that I and O must include a meta-morphism. Therefore, unsafe communication is often used in practice, and even that can be difficult to implement.

Our framework is less appropriate if $\mathcal{S}_1$ or $\mathcal{S}_2$ are developed using the definitional method. For example, consider Aczel's encoding of set theory in type theory [Acz98, Wer97]. Here $\mathcal{S}_1 = \mathbf{Nat}$ as in Ex. 2, and $\mathcal{S}_2 = \texttt{Nat}$ as in Ex. 1. Aczel's encoding provides the needed meta-morphism $l : \mathbf{ZF} \to \texttt{CIC}$ of O. But because **Nat** is definitional, we already have $O = l$, and we have no freedom to define O such that it maps the concepts of **Nat** to their counterparts in `Nat`. Formally, in MMT, this means that the condition $O \circ \mu_1 = \mu_2$ fails. Instead, we obtain two versions of the natural numbers in CIC: a native one given by μ_2 and the translation of **Nat** given by $O \circ \mu_1$. Indeed, the latter must satisfy all **ZF**-theorems including, e.g., $0 \in 1$, which is not even a well-formed formula over `Nat`. We speak of *faithful communication* if $O \circ \mu_1 = \mu_2$ can be established even when $\mathcal{S}_1$ is definitional. This is not possible in MMT without the extension we propose below.

4 A Framework for System Integration

In order to realize faithful communication within MMT, we introduce *partial theory morphisms* that can filter out those definitional details of $\mathcal{S}_1$ that need not and cannot be mapped to $\mathcal{S}_2$. We will develop this new concept in general in Sect. 4.1 and then apply it to the integration problem in Sect. 4.2.

4.1 Partial Theory Morphisms in MMT

Syntax. We extend the MMT syntax with the production $O ::= \top$. The intended use of $\top$ is to put assignments $c \mapsto \top$ into the body of a morphism $v : S \to$

$T \overset{l}{=} \{\sigma\}$ in order to make v undefined at c. We say that v *filters* c. The homomorphic extension $v(-)$ remains unchanged and is still total: If O contains filtered symbols, then $v(O)$ contains $\top$ as a subobject. In that case, we say v *filters* O.

Semantics. We refine the semantics as follows. A *dependency cut* D for an MMT theory T is a pair (D_{type}, D_{def}) of two sets of symbols accessible to T. Given such a dependency cut, we define *dependency-aware judgments* $\gamma \vdash_D O : O'$ and $\gamma \vdash_D O = O'$ as follows. $\gamma \vdash_D O : O'$ means that there is a derivation of $\gamma \vdash_T O : O'$ that uses the rules $\mathcal{T}_:$ and $\mathcal{T}_=$ at most for the constants in D_{type} and D_{def}, respectively. $\gamma \vdash_D O = O'$ is defined accordingly.

In other words, if we have $\gamma' \vdash_D O : O'$ and obtain γ' by changing the type of any constant not in D_{type} or the definiens of any constant not in D_{def}, then we still have $\gamma' \vdash_D O : O'$. Then a *foundation* consists of a foundational theory L together with dependency-aware judgments for typing and equality whenever T has meta-theory L.

We make a crucial change to the MMT rule for assignments in a theory morphism: If S contains a declaration $c : O_1 = O_2$, then a theory morphism $v : S \to T \overset{l}{=} \{\sigma\}$ may contain the assignment $c \mapsto O$ only if the following two conditions hold: (i) if O_1 is not filtered by v, then $\vdash_T O : v(O_1)$; (ii) if O_2 is not filtered by v, then $\vdash_T O = v(O_2)$. The according rule applies if O_1 or O_2 are omitted.

In [RK11], a stricter condition is used. There, if O_1 or O_2 are filtered, then c must be filtered as well. While this is a natural strictness condition for filtering, it is inappropriate for our use cases: For example, filtering all L-symbols would entail filtering all $\mathcal{S}$-symbols.

Our weakened strictness condition is still strong enough to prove the central property of theory morphisms: If $\gamma \vdash \mu : S \to T$ and $\vdash_D O : O'$ for some $D = (D_{type}, D_{def})$ and v does not filter O, O', the type of a constant in D_{type}, or the definiens of a constant in D_{def}, then $\vdash_T \mu(O) : \mu(O')$. The according result holds for the equality judgment.

Finally, we define the *weak equality of morphisms* $\mu_i : S \to T$. We define $\vdash \mu_1 \leq \mu_2$ in the same way as $\vdash \mu_1 = \mu_2$ except that $\vdash_T \mu_1(c) = \mu_2(c)$ is only required if c is not filtered by μ_1. We say that $\vdash \eta : T \to S$ is a *partial inverse* of $\mu : S \to T$ if $\vdash \eta \circ \mu = id_S$ and $\vdash \mu \circ \eta \leq id_T$.

Example 3. Consider the morphism $\mu_1 : Nat \to \mathbf{Nat}$ from Ex. 2. We build its partial inverse $\eta : \mathbf{Nat} \to Nat \overset{l}{=} \{\sigma\}$. The meta-morphism l filters all symbols of **ZF**, e.g., $l(\varnothing) = \top$. Then the symbol $\mathbf{N}$ of **Nat** has filtered type and filtered definiens. Therefore, the conditions (i) and (ii) above are vacuous, and we use $\mathbf{N} \mapsto N$ in σ. Then all remaining symbols of **Nat** (including the theorems) have filtered definiens but unfiltered types. For example, for $\mathbf{0} : \mathbf{N} = \varnothing$ we have $\eta(\varnothing) = \top$ but $\eta(\mathbf{N}) = N$. Therefore, condition (ii) is vacuous, and we map these symbols to their counterparts in Nat, e.g., using $\mathbf{0} \mapsto 0$ in σ. These assignments are type-preserving as required by condition (i) above, e.g., $\vdash_{Nat} \eta(\mathbf{0}) : \eta(\mathbf{N})$.

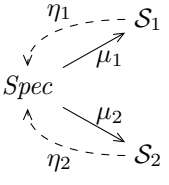
4.2 Integration via Partial Theory Morphisms

The following gives a typical application of our framework by safely and faithfully communicating proofs from a stronger to a weaker system:

Example 4. In [IR11], we gave formalizations of Zermelo-Fraenkel (ZFC) set theory and Mizar's Tarski-Grothendieck set theory (TG) using the logical framework LF as the common meta-theory. ZFC and TG share the language of first-order set theory. But TG is stronger than ZFC because of Tarski's axiom, which implies, e.g, the sentence I stating the existence of infinite sets (which is an axiom in ZFC) and large cardinals (which is unprovable in ZFC). For example, we have an axiom $a_\infty : I$ in ZFC, and an axiom $tarski : T$ and a theorem $t_\infty : I = P$ in TG. Many TG-theorems do not actually depend on this additional strength, but they do depend on t_∞ and thus indirectly on $tarski$.

Using our framework, we can capture such a theorem as the case of a TG-theorem $\vdash_D p : F$ where F is the theorem statement and $t_\infty \in D_{type}$ but $t_\infty \notin D_{def}$ and $tarski \notin D_{type}$. We can give a partial theory morphism $v : TG \to ZFC \stackrel{id_{LF}}{=} \{\ldots, t_\infty \mapsto a_\infty, \ldots\}$. Then v does not filter p, and we obtain $\vdash_{ZFC} v(p) : F$.

Assume now that we have two implementations $\mu_i : Spec \to \mathcal{S}_i$ of $Spec$ and partial inverses η_i of μ_i, where $\mathcal{S}_i$ has meta-theory L_i. This leads to the diagram on the right where (dashed) edges are (partial) theory morphisms. We can now obtain the translations $I : \mathcal{S}_2 \to \mathcal{S}_1$ and $O : \mathcal{S}_1 \to \mathcal{S}_2$ as $I = \mu_1 \circ \eta_2$ and $O = \mu_2 \circ \eta_1$. Note that I and O are partial inverses of each other.



As in Sect. 3, let $C \vdash_2 ? : F$ be a query in $\mathcal{S}_2$. If η_2 does not filter any symbols in C or F, we obtain the translated problem $I(C) \vdash_1 ? : I(F)$. Let us further assume that there is an $\mathcal{S}_1$-substitution $\vdash_1 s : I(C)$ and a proof $\vdash_1 p : I(F)[s]$ such that p and s are not filtered by η_1. Because I and O are mutually inverse and morphism application preserves typing, we obtain the solution $\vdash_2 O(p) : F[O(s)]$.

The condition that η_2 does not filter C and F is quite reasonable in practice: Otherwise, the meaning of the query would depend on implementation-specific details of $\mathcal{S}_2$, and it is unlikely that $\mathcal{S}_1$ should be able to find an answer anyway. On the other hand, the morphism η_1 is more likely to filter the proof p. Moreover, since the proof must be translated from L_1 to L_2 passing through $Spec$, the latter must include a proof system to allow translation of proofs. In practice this is rarely the case, even if the consequence relation of $Spec$ can be expressed as an inference system. For example, large parts of mathematics or the OpenMath content dictionaries implicitly (import) first-order logic and ZF set theory.

We outline two ways how to remedy this: We can *communicate filtered proofs* or change the morphisms to *widen the filters* to let more proofs pass.

Communicating Filtered Proofs. Firstly, if the proof rules of $\mathcal{S}_1$ are filtered by η_1, what is received by $\mathcal{S}_2$ after applying the output translation O is a filtered proof, i.e., a proof object that contains the constant $\top$. $\top$ represents gaps in the proof that were lost in the translation.

In an extreme case, all applications of proof rules become $\top$, and the only unfiltered parts of $O(p)$ are formulas that occurred as intermediate results during the proof. In that case, $O(p)$ is essentially a list of formulas F_i (a proof sketch in the sense of [Wie03]) such that $I(F_1) \wedge \ldots \wedge I(F_{i-1}) \vdash_1 I(F_i)$ for $i = 1, \ldots, n$. In order to refine $O(p)$ into a proof, we have to derive $\vdash_1 F_n$. Most of the time, it will be the case that $F_1, \ldots, F_{i-1} \vdash_2 F_i$ for all i, and the proof is obtained compositionally if $\mathcal{S}_2$ can fill the gaps through automated reasoning. When this happens, the proof sketch is already a complete declarative proof.

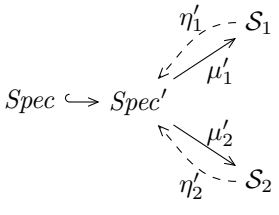
Example 5. Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be implementations of the rational numbers with different choices for division by zero. In $\mathcal{S}_1$, division by zero yields a special value for undefined results, and operations on undefined values yield undefined results; then we have the $\mathcal{S}_1$-theorem t asserting $\forall a, b, c.a(b/c) \doteq (ab)/c$. In $\mathcal{S}_2$, we have $n/0 \doteq 1$ and $n\%0 \doteq n$; then we have the $\mathcal{S}_2$-theorems t_1, t_2, t_3 asserting $\forall m, n.n \doteq (n/m) * m + n\%m$, $\forall m.m/m \doteq 1$, and $\forall m.m\%m \doteq 0$.

The choice in $\mathcal{S}_2$ reduces the number of case analyses in basic proofs. But t is not a theorem of $\mathcal{S}_2$; instead, we only have a theorem t' asserting $\forall a, b, c.c \not\doteq 0 \Rightarrow a(b/c) \doteq (ab)/c$. On the other hand, $\mathcal{S}_1$ is closer to common mathematics, but the t_i are not theorems of $\mathcal{S}_1$ because the side condition $m \neq 0$ is needed.

Hence, we do not have a total theory morphism $O : \mathcal{S}_1 \to \mathcal{S}_2$, but we can give a partial theory morphism O that filters t. Now consider, for example, a proof p over $\mathcal{S}_1$ that instantiates t with some values A, B, C. When translating p to $\mathcal{S}_2$, t is filtered, but we can still communicate p, and $\mathcal{S}_2$ can treat $O(p)$ as a proof sketch. Typically, t is applied in a context where $C \not\doteq 0$ is known anyway so that $\mathcal{S}_2$ can patch $O(p)$ by using t' — which can easily be found by automated reasoning.

Integration in the other direction works accordingly.

Widening the Filters. An alternative solution is to use additional knowledge about $\mathcal{S}_1$ and $\mathcal{S}_2$ to obtain a translation where $O(p)$ is not filtered. In particular, if p is filtered completely, we can strengthen *Spec* by adding an inference system for the consequence relation of *Spec*, thus obtaining *Spec*$'$. Then we can extend the morphisms μ_i accordingly to μ_i', which amounts to proving that $\mathcal{S}_i$ is a correct implementation of *Spec*. Now η_i can be extended as well so that its domain becomes bigger, i.e., the morphism η_1 and thus O filter less proofs and become "wider".



Note that we are flexible in defining *Spec*$'$ as required by the particular choices of L_1 and L_2. That way the official specification remains unchanged, and we can maximize the filters for every individual integration scenario.

Example 6 (Continuing Ex. 3). A typical situation is that we have a theorem F over **Nat** whose proof p uses the Peano axioms and the rules of first-order logic but does not expand the definitions of the natural numbers. Moreover, if $a : A = P$ is a theorem in **Nat** that establishes one of the Peano axioms, then p

will refer to a, but will not expand the definition of a. Formally, we can describe this as $\vdash_D p : F$ where $\mathbf{0}, a \in D_{type}$ but $\mathbf{0}, a \notin D_{def}$.

We can form $Spec'$ by extending $Spec$ with proof rules for first-order logic and extend η to η' accordingly. Since η does not filter the types of $\mathbf{0}$ and a, we obtain a proof $\vdash_{Spec} \eta'(p) : \eta'(F)$ due to the type-preservation properties of our partial theory morphisms. Despite the partiality of η', the correctness of this proof is guaranteed by the framework.

Both ways to integrate systems are not new and have been used ad hoc in concrete integration approaches, see Sect. 5. With our framework, we are able to capture them in a rigorous framework where their soundness can be studied formally.

5 Related Work

The MoWGLI project [MoW04] introduced the concept of "semantic markup" for specifications in the calculus of construction as distinct from the "content markup" in OpenMath and OMDoc. This corresponds closely to the use of meta-theories in MMT: "content markup" corresponds to MMT theories without meta-theory; and "semantic markup" corresponds to MMT theories with meta-theory CIC.

A framework very similar to ours was given in [CFW03]. Our MMT theories with meta-theory correspond to their biform theories, except that the latter adds algorithms. Our theory morphisms I and O correspond to their translations `export` and `import`. The key improvement of our framework over [CFW03] is that, using MMT's meta-theories, the involved logics and their consequence relations can be defined declaratively themselves so that a logic-independent implementation becomes possible. Similarly, using logic morphisms, it becomes possible to implement and verify the trustability conditions concisely.

Integration by borrowing is the typical scenario of integrating theorem provers and proof assistants. For example, Leo-II [BPTF08] or the Sledgehammer tactic of Isabelle [MP08] ($\mathcal{S}_2$) use first-order provers ($\mathcal{S}_1$) to reason in higher-order logic. Here the input translation I is partial inverse of the inclusion from first-order logic to higher-order logic. A total translation from modal logic to first-order logic is used in [HS00]. In all cases, the safety is verified informally on the meta-level and no output translation O in our sense is used. But Isabelle makes the communication safe by reconstructing a proof from the proof (sketch) returned by the prover.

The above systems are called on demand using an input translation I. Alternatively a collection of $\mathcal{S}_1$-proofs can be translated via an output translation O for later reuse in $\mathcal{S}_2$; in that case no input translation I is used at all. Examples are the translations from Isabelle/HOL in HOL Light [McL06], from HOL Light to Isabelle/HOL [OS06], from HOL Light to Coq [KW10], or from Isabelle/HOL to Isabelle/ZF [KS10]. The translation from HOL to Isabelle/HOL is notable because it permits faithful translations, e.g., the real numbers of HOL can be translated to the real numbers of Isabelle/HOL, even though the two systems

define them differently. The safety of the translation is achieved by recording individual $\mathcal{S}_1$-proofs and replaying them in $\mathcal{S}_2$. This was difficult to achieve even though $\mathcal{S}_1$ and $\mathcal{S}_2$ are based on the same logic.

The translation given in [KW10] is the first faithful translation from HOL proofs to CIC proofs. Since the two logics are different, in order to obtain a total map the authors widen the filter by assuming additional axioms on CIC (excluded middle and extensionality of functions). This technique is not exploitable when the required axioms are inconsistent. Moreover, the translation is suboptimal, since it uses excluded middle also for proofs that are intuitionistic. To improve the solution, we could use partial theory morphisms that map case analysis over boolean in HOL to $\top$, and then use automation to avoid excluded middle in CIC when the properties involved are all decidable.

In all above examples but [KW10], the used translations are not verified within a logical framework. The Logosphere [PSK+03] project used the proof theoretical framework LF to provide statically verified logic translations that permit inherently safe communication. Here the dynamic verification of translated proofs becomes redundant. The most advanced such proof translation is one from HOL to Nuprl [NSM01].

The theory of institutions [GB92] provides a general model theoretical framework in which borrowing has been studied extensively [CM97] and implemented successfully [MML07]. Here the focus is on giving the morphism I explicitly and using a model theoretical argument to establish the existence of some O; then communication is safe without explicitly translating proofs.

Integration by computation is the typical scenario for the integration of computer algebra systems, which is the main topic of the Calculemus series of conferences. For typical examples, see [DM05] where the computation is performed by a CAS, and [AT07] where the computation is done by a term rewriting system. Communication is typically unsafe. Alternatively, safety can be achieved if the results of the CAS — e.g., the factorization of a polynomial — can be verified formally in a DS as done in [HT98] and [Sor00].

Typical applications of integration by querying are conjunctive query answering for a description logic. For example, in [TSP08], a first-order theorem prover is used to answer queries about the SUMO ontology.

The communication of filtered proofs essentially leads to formal proof sketch in the sense of [Wie03]. The idea of abstracting from a proof to a proof sketch corresponds to the assertion level proofs used in [Mei00] to integrate first-order provers. The recording and replaying of proof steps in [OS06] and the reconstruction of proofs in Isabelle are also special cases of the communication of filtered proofs.

6 Conclusion

In this paper we addressed the problem of preserving the semantics in protocol-based integration of mathematical reasoning and computation systems. We analyzed the problem from a foundational point of view and proposed a framework

based on theory graphs, partial theory morphisms, and explicit representations of meta-logics that allows to state solutions to the integration problem.

The main contribution and novelty of the paper is that it paves the way towards a *theory of integration*. Theoretically, via filtering, this theory could be able to combine faithfulness with static verification, which would be a major step towards the integration and merging of system libraries. Moreover, we believe it is practical because it requires only a simple extension of the MMT framework, which already takes scalability issues very seriously [KRZ10].

We do not expect that our specific solution covers all integration problems that come up in practice. But we do expect that it will take a long time to exhaust the potential that our framework offers.

References

[Acz98] Aczel, P.: On relating type theories and set theories. In: Altenkirch, T., Naraschewski, W., Reus, B. (eds.) TYPES 1998. LNCS, vol. 1657, pp. 1–18. Springer, Heidelberg (1999)

[AT07] Asperti, A., Tassi, E.: Higher order Proof Reconstruction from Paramodulation-Based Refutations: The Unit Equality Case. In: Kauers, M., Kerber, M., Miner, R., Windsteiger, W. (eds.) MKM/CALCULEMUS 2007. LNCS (LNAI), vol. 4573, pp. 146–160. Springer, Heidelberg (2007)

[BC04] Bertot, Y., Castéran, P.: Coq'Art: The Calculus of Inductive Constructions. Springer, Heidelberg (2004)

[BPTF08] Benzmüller, C., Paulson, L., Theiss, F., Fietzke, A.: LEO-II - A Cooperative Automatic Theorem Prover for Classical Higher-Order Logic (System Description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 162–170. Springer, Heidelberg (2008)

[CF58] Curry, H., Feys, R.: Combinatory Logic. North-Holland, Amsterdam (1958)

[CFW03] Carette, J., Farmer, W., Wajs, J.: Trustable Communication between Mathematics Systems. In: Hardin, T., Rioboo, R. (eds.) Proceedings of Calculemus, pp. 58–68 (2003)

[CM97] Cerioli, M., Meseguer, J.: I Borrow Your Logic (Transporting Logical Structures along Maps). Theoretical Computer Science 173, 311–347 (1997)

[Del99] Delahaye, D.: Information Retrieval in a Coq Proof Library Using Type Isomorphisms. In: Coquand, T., Nordström, B., Dybjer, P., Smith, J. (eds.) TYPES 1999. LNCS, vol. 1956, pp. 131–147. Springer, Heidelberg (2000)

[DM05] Delahaye, D., Mayero, M.: Dealing with Algebraic Expressions over a Field in Coq using Maple. Journal of Symbolic Computation 39(5), 569–592 (2005)

[Far04] Farmer, W.: Formalizing Undefinedness Arising in Calculus. In: Basin, D., Rusinowitch, M. (eds.) IJCAR 2004. LNCS (LNAI), vol. 3097, pp. 475–489. Springer, Heidelberg (2004)

[GB92] Goguen, J., Burstall, R.: Institutions: Abstract model theory for specification and programming. Journal of the Association for Computing Machinery 39(1), 95–146 (1992)

[HHP93] Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. Journal of the Association for Computing Machinery 40(1), 143–184 (1993)

[How80] Howard, W.: The formulas-as-types notion of construction. In: To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism, pp. 479–490. Academic Press, London (1980)

[HS00] Hustadt, U., Schmidt, R.: MSPASS: Modal Reasoning by Translation and First-Order Resolution. In: Dyckhoff, R. (ed.) TABLEAUX 2000. LNCS, vol. 1847, pp. 67–71. Springer, Heidelberg (2000)

[HT98] Harrison, J., Théry, L.: A Skeptic's Approach to Combining HOL and Maple. Journal of Automated Reasoning 21, 279–294 (1998)

[IR11] Iancu, M., Rabe, F.: Formalizing Foundations of Mathematics. Mathematical Structures in Computer Science (to appear 2011), http://kwarc.info/frabe/Research/IR_foundations_10.pdf

[KRZ10] Kohlhase, M., Rabe, F., Zholudev, V.: Towards MKM in the Large: Modular Representation and Scalable Software Architecture. In: Autexier, S., Calmet, J., Delahaye, D., Ion, P., Rideau, L., Rioboo, R., Sexton, A. (eds.) AISC 2010. LNCS, vol. 6167, pp. 370–384. Springer, Heidelberg (2010)

[KS10] Krauss, A., Schropp, A.: A Mechanized Translation from Higher-Order Logic to Set Theory. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 323–338. Springer, Heidelberg (2010)

[KW10] Keller, C., Werner, B.: Importing HOL Light into Coq. In: Kaufmann, M., Paulson, L. (eds.) ITP 2010. LNCS, vol. 6172, pp. 307–322. Springer, Heidelberg (2010)

[McL06] McLaughlin, S.: An Interpretation of Isabelle/HOL in HOL Light. In: Shankar, N., Furbach, U. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 192–204. Springer, Heidelberg (2006)

[Mei00] Meier, A.: System Description: TRAMP: Transformation of Machine-Found Proofs into ND-Proofs at the Assertion Level. In: McAllester, D. (ed.) CADE 2000. LNCS, vol. 1831, pp. 460–464. Springer, Heidelberg (2000)

[MML07] Mossakowski, T., Maeder, C., Lüttich, K.: The Heterogeneous Tool Set. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 519–522. Springer, Heidelberg (2007)

[MoW04] MoWGLI Project Deliverables (2004), http://mowgli.cs.unibo.it/html_no_frames/deliverables/index.html

[MP08] Meng, J., Paulson, L.: Translating Higher-Order Clauses to First-Order Clauses. Journal of Automated Reasoning 40(1), 35–60 (2008)

[NSM01] Naumov, P., Stehr, M., Meseguer, J.: The hOL/NuPRL proof translator (A practical approach to formal interoperability). In: Boulton, R.J., Jackson, P.B. (eds.) TPHOLs 2001. LNCS, vol. 2152, pp. 329–345. Springer, Heidelberg (2001)

[OS06] Obua, S., Skalberg, S.: Importing HOL into Isabelle/HOL. In: Shankar, N., Furbach, U. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 298–302. Springer, Heidelberg (2006)

[PSK+03] Pfenning, F., Schürmann, C., Kohlhase, M., Shankar, N., Owre, S.: The Logosphere Project (2003), http://www.logosphere.org/

[RK11] Rabe, F., Kohlhase, M.: A Scalable Module System, (2011), http://arxiv.org/abs/1105.0548 (under review)

[Sor00] Sorge, V.: Non-trivial Symbolic Computations in Proof Planning. In: Kirchner, H. (ed.) FroCos 2000. LNCS, vol. 1794, pp. 121–135. Springer, Heidelberg (2000)

[TSP08] Trac, S., Sutcliffe, G., Pease, A.: Integration of the TPTPWorld into SigmaKEE. In: Konev, B., Schmidt, R., Schulz, S. (eds.) Practical Aspects of Automated Reasoning. CEUR Workshop Proceedings, vol. 373 (2008)

[Wer97] Werner, B.: Sets in types, types in sets. In: Abadi, M., Ito, T. (eds.) TACS 1997. LNCS, vol. 1281, pp. 530–546. Springer, Heidelberg (1997)

[Wie03] Wiedijk, F.: Formal Proof Sketches. In: Berardi, S., Coppo, M., Damiani, F. (eds.) TYPES 2003. LNCS, vol. 3085, pp. 378–393. Springer, Heidelberg (2004)

Efficient Formal Verification of Bounds of Linear Programs

Alexey Solovyev and Thomas C. Hales*

Department of Mathematics, University of Pittsburgh,
Pittsburgh, PA 15260, USA

Abstract. One of the challenging problems in the formalization of mathematics is a formal verification of numerical computations. Many theorems rely on numerical results, the verification of which is necessary for producing complete formal proofs. The formal verification systems are not well suited for doing high-performance computing since even a small arithmetic step must be completely justified using elementary rules. We have developed a set of procedures in the HOL Light proof assistant for efficient verification of bounds of relatively large linear programs. The main motivation for the development of our tool was the work on the formal proof of the Kepler Conjecture. An important part of the proof consists of about 50000 linear programs each of which contains more than 1000 variables and constraints. Our tool is capable to verify one such a linear program in about 5 seconds. This is sufficiently fast for doing the needed formal computations.

1 Introduction

A trivial arithmetic step is not a problem in a traditional mathematical proof. The situation is different when one needs to obtain a formal proof where each step requires a complete verification using definitions, elementary logic operations, and axioms. Nevertheless, most proof assistant systems have special automated procedures that are able to prove simple arithmetic and logical statements. The main drawback of universal procedures is that they are not intended for high-performance computations. There exist theorems which require substantial computations. The formalization of these theorems can be quite challenging. One example is the four-color theorem, which was successfully formalized in Coq [1]. Another ambitious formalization project is the Flyspeck project [2]. The goal of this project is the formal proof of the Kepler conjecture [3]. This proof relies on extensive computer computations. An important part of the proof consists of more than 50000 linear programs (with about 1000 variables and constraints each). A bound of each linear program needs to be formally verified.

In our work, we present a tool for proving bounds of linear programs. The hard computational work is done using external software for solving linear programs. This software returns a special certificate which is used in the formal verification procedure. There are two main difficulties in this approach. One is the

* Research supported by NSF grant 0804189 and a grant from the Benter Foundation.

J.H. Davenport et al. (Eds.): Calculemus/MKM 2011, LNAI 6824, pp. 123–132, 2011.

precision of the computer arithmetic. Usually, results of computer floating-point operations are not exact. Meanwhile, precise results are necessary for producing formal proofs. The second problem is the speed of the formal arithmetic. We successfully solved both problems and our tool is able to verify relatively large Flyspeck linear programs in about 5 seconds.

The Flyspeck project is carried out in the HOL Light proof assistant [5]. HOL Light is written in the Objective CAML programming language [10]. It has many convenient automated tools for proving arithmetic statements and can prove bounds of linear programs automatically. But even for small linear programs (10 variables and less) it can take a lot of time. Steven Obua in his thesis [8] developed a tool for verifying a part of Flyspeck linear programs. His work is done in the Isabelle proof assistant. We have made three significant advances over this earlier work. First, the combinatorial aspects of the linear programs have been simplified, as described in [4]. Second, we developed a tool for verifying bounds of general linear programs which verifies Flyspeck linear program much faster than Obua's program (in his work, the time of verification of a single linear program varies from 8.4 minutes up to 67 minutes). Third, our work is done in HOL Light so it is not required to translate results from one proof assistant into another. The code of our tool can be found in the Flyspeck repository [2] at trunk/formal_lp.

2 Verification of Bounds of Linear Programs

Our goal is to prove inequalities in the form $\mathbf{c}^T\mathbf{x} \le K$ such that $\mathbf{A}\mathbf{x} \le \mathbf{b}$ and $\mathbf{l} \le \mathbf{x} \le \mathbf{u}$, where $\mathbf{c}$, $\mathbf{b}$, $\mathbf{l}$, $\mathbf{u}$ are given n-dimensional vectors, $\mathbf{x}$ is an n-dimensional vector of variables, K is a constant, and $\mathbf{A}$ is an $m \times n$ matrix. To solve this problem, we consider the following linear program

$$\text{maximize } \mathbf{c}^T\mathbf{x} \text{ subject to } \bar{\mathbf{A}}\mathbf{x} \le \bar{\mathbf{b}},\ \bar{\mathbf{A}} = \begin{pmatrix} \mathbf{A} \\ -\mathbf{I}_n \\ \mathbf{I}_n \end{pmatrix},\ \bar{\mathbf{b}} = \begin{pmatrix} \mathbf{b} \\ -\mathbf{l} \\ \mathbf{u} \end{pmatrix}.$$

Suppose that $M = \max \mathbf{c}^T\mathbf{x}$ is the solution to this linear program. We require that $M \le K$. In fact, for our method we need a strict inequality $M < K$ because we employ numerical methods which do not give exact solutions.

We do not want to solve the linear program given above using formal methods. Instead, we use general software for solving linear programs which produces a special certificate that can be used to formally verify the original upper bound. Consider a dual linear program

$$\text{minimize } \mathbf{y}^T\bar{\mathbf{b}} \text{ subject to } \mathbf{y}^T\bar{\mathbf{A}} = \mathbf{c}^T,\ \mathbf{y} \ge 0.$$

The general theory of linear programming asserts that if the primal linear program has an optimal solution, then the dual program also has an optimal solution such that $\min \mathbf{y}^T\bar{\mathbf{b}} = \max \mathbf{c}^T\mathbf{x} = M$. Suppose that we can find an optimal solution to the dual program, i.e., assume that we know $\mathbf{y}$ such that $\mathbf{y}^T\bar{\mathbf{b}} = M \le K$

and $\mathbf{y}^T\bar{\mathbf{A}} = \mathbf{c}^T$. Then we can formally verify the original inequality by doing the following computations in a formal way:

$$\mathbf{c}^T\mathbf{x} = (\mathbf{y}^T\bar{\mathbf{A}})\mathbf{x} = \mathbf{y}^T(\bar{\mathbf{A}}\mathbf{x}) \leq \mathbf{y}^T\bar{\mathbf{b}} = M \leq K.$$

Our algorithm can be split into two parts. In the first part, we compute a solution $\mathbf{y}$ to the dual problem. In the second part, we formally prove the initial inequality using the computed dual solution and doing all arithmetic operations in a formal way.

2.1 Finding a Dual Solution

We impose additional constraints on the input data. We suppose that all coefficients and constants can be approximated by finite decimal numbers such that a solution of the approximated problem implies the original inequality. Consider a simple example. Suppose we need to prove the inequality $x - y \leq \sqrt{3}$ subject to $0 \leq x \leq \pi$ and $\sqrt{2} \leq y \leq 2$. In general, an approximation that loosens the domain and tightens the range implies the original inequality. For example, consider an approximation of proving $x - y \leq 1.732$, subject to $0 \leq x \leq 3.142$ and $1.414 \leq y \leq 2$. It is easy to see that if we can prove the approximated inequality, then the verification of the original inequality trivially follows. From now on, we assume that entries of $\bar{\mathbf{A}}$, $\bar{\mathbf{b}}$, $\mathbf{c}$, and the constant K are finite decimal numbers with at most p_1 decimal digits after the decimal point.

We need to find a vector $\mathbf{y}$ with the following properties:

$$\mathbf{y} \geq 0, \quad \mathbf{y}^T\bar{\mathbf{A}} = \mathbf{c}^T, \quad \mathbf{y}^T\bar{\mathbf{b}} \leq K.$$

Moreover, we require that all elements of $\mathbf{y}$ are finite decimal numbers.

In our work, we use `GLPK` (GNU Linear Programming Kit) software for solving linear programs [7]. The input of this program is a model file which describes a linear program in the AMPL modeling language [9]. `GLPK` automatically finds solutions of the primal and dual linear programs. We are interested in the dual solution only. Suppose $\mathbf{r}$ is a numerical solution to the dual problem. Take its decimal approximation $\mathbf{y}_1^{(p)}$ with p decimal digits after the decimal point. We have the following properties of $\mathbf{y}_1^{(p)}$:

$$\mathbf{y}_1^{(p)} \geq 0, \quad M \leq \bar{\mathbf{b}}^T\mathbf{y}_1^{(p)}, \quad \bar{\mathbf{A}}^T\mathbf{y}_1^{(p)} = \mathbf{c} + \mathbf{e}.$$

The vector $\mathbf{e}$ is the error term from numerical computation and decimal approximation.

We need to modify the numerical solution $\mathbf{y}_1^{(p)}$ to get $\mathbf{y}_2^{(p)}$ such that $\bar{\mathbf{A}}^T\mathbf{y}_2^{(p)} = \mathbf{c}$. Write $\mathbf{y}_1^{(p)} = (\mathbf{z}^T, \mathbf{v}^T, \mathbf{w}^T)^T$ where $\mathbf{z}$ is an m-dimensional vector, $\mathbf{v}$ and $\mathbf{w}$ are n-dimensional vectors. Define $\mathbf{y}_2^{(p)}$ as follows

$$\mathbf{y}_2^{(p)} = \begin{pmatrix} \mathbf{z} \\ \mathbf{v} + \mathbf{v}_e \\ \mathbf{w} + \mathbf{w}_e \end{pmatrix}, \quad \mathbf{v}_e = \frac{|\mathbf{e}| + \mathbf{e}}{2}, \quad \mathbf{w}_e = \frac{|\mathbf{e}| - \mathbf{e}}{2}.$$

In other words, if $e_i > 0$ (the i-th component of $\mathbf{e}$), then we add e_i to v_i, otherwise we add $-e_i$ to w_i. We obtain $\mathbf{y}_2^{(p)} \geq 0$. Moreover,

$$\bar{\mathbf{A}}^T \mathbf{y}_2^{(p)} = \mathbf{A}^T \mathbf{z} - (\mathbf{v} + \mathbf{v}_e) + (\mathbf{w} + \mathbf{w}_e) = \bar{\mathbf{A}}^T \mathbf{y}_1^{(p)} - \mathbf{e} = \mathbf{c}.$$

Note that elements of $\mathbf{y}_2^{(p)}$ are finite decimal numbers. Indeed, $\mathbf{y}_2^{(p)}$ is obtained by adding some components of the error vector $\mathbf{e} = \bar{\mathbf{A}}^T \mathbf{y}_1^{(p)} - \mathbf{c}$ to the vector $\mathbf{y}_1^{(p)}$, and all components of $\bar{\mathbf{A}}$, $\mathbf{c}$, and $\mathbf{y}_1^{(p)}$ are finite decimal numbers.

If $\bar{\mathbf{b}}^T \mathbf{y}_2^{(p)} \leq K$, then we are done. Otherwise, we need to find $\mathbf{y}_1^{(p+1)}$ using higher precision decimal approximation of $\mathbf{r}$ and consider $\mathbf{y}_2^{(p+1)}$. Assuming that the numerical solution $\mathbf{r}$ can be computed with arbitrary precision and that $M < K$, we eventually get $\bar{\mathbf{b}}^T \mathbf{y}_2^{(s)} \leq K$.

From the computational point of view, we are interested in finding an approximation of the dual solution such that its components have as few decimal digits as possible (formal arithmetic on small numbers works faster). We start from a small value of p_0 (we choose $p_0 = 3$ for Flyspeck linear programs) and construct $\mathbf{y}_2^{(p_0)}, \mathbf{y}_2^{(p_0+1)}, \ldots$ until we get $\bar{\mathbf{b}}^T \mathbf{y}_2^{(p_0+i)} \leq K$.

We implemented a program in C# which takes a model file with all inequalities and a dual solution obtained with `GLPK`. The program returns an approximate dual solution (with as low precision as possible) which then can be used in the formal verification step. The current implementation of our program does not work with arbitrary precision arithmetic, so it could fail on some linear programs. It should be not a problem for many practical cases because standard double precision floating-point arithmetic can exactly represent 15-digit decimal numbers (for instance, we need at most 6 decimal digits for proving Flyspeck linear programs).

2.2 Formal Verification

Our aim is to verify the inequality $\mathbf{c}^T \mathbf{x} \leq K$ using the computed dual solution approximation $\mathbf{y}^T = (\mathbf{z}^T, \mathbf{v}^T, \mathbf{w}^T)$ (we write $\mathbf{y}$ for the approximation $\mathbf{y}_2^{(s)}$, computation of which is described in the previous section):

$$\mathbf{c}^T \mathbf{x} = \mathbf{z}^T (\mathbf{A}\mathbf{x}) - \mathbf{v}^T \mathbf{x} + \mathbf{w}^T \mathbf{x} = \mathbf{y}^T \bar{\mathbf{A}} \mathbf{x} \leq \mathbf{y}^T \bar{\mathbf{b}} \leq K.$$

Here $\mathbf{x}$ is an n-dimensional vector of variables, $\mathbf{x} = (x_1, \ldots, x_n)$. We need to verify two results using formal arithmetic: $\mathbf{y}^T \bar{\mathbf{A}} \mathbf{x} = \mathbf{c}^T \mathbf{x}$ and $\mathbf{y}^T \bar{\mathbf{b}} \leq K$.

The computation of $\mathbf{y}^T \bar{\mathbf{b}}$ is a straightforward application of formal arithmetic operations. $\mathbf{y}^T \bar{\mathbf{A}} \mathbf{x}$ can be computed in a quite efficient way. Usually, the matrix $\bar{\mathbf{A}}$ is sparse, so it makes no sense to do a complete matrix multiplication in order to compute $\mathbf{y}^T \bar{\mathbf{A}} \mathbf{x}$. The i-th constraint inequality can be written in the form

$$\sum_{j \in I_i} a_{ij} x_j \leq b_i,$$

where I_i is the set of indices such that $a_{ij} \neq 0$ for $j \in I_i$, and $a_{ij} = 0$ for $j \notin I_i$. Also we have $2n$ inequalities for bounds of $\mathbf{x}$: $l_i \leq x_i \leq u_i$.

Define a special function in HOL Light for representing the left hand side of a constraint inequality

$$\vdash \text{linf}[] = \&0 \wedge \text{linf}[(a_1, x_1); t] = a_1 * x_1 + \text{linf}[t]$$

For the sake of presentation, we write HOL Light expressions in a simplified notation. Here, $\vdash$ means that the definition is a HOL Light theorem; &0 denotes the real number zero. Our function linf has the following type

$$\text{linf} : (real, real)list \to real.$$

It means that linf takes a list of pairs of real-valued elements and returns a real value. The function linf is defined recursively: we define its value on the empty list [], and we specify how linf can be computed on a k-element list using its value on a $(k-1)$-element list.

We suppose that all constraints and bounds of variables are theorems in HOL Light and each such theorem has the form

$$\vdash \alpha_1 x_1 + \ldots + \alpha_k x_k \leq \beta$$

We have a conversion which transforms $\alpha_1 x_1 + \ldots + \alpha_k x_k$ into the corresponding function $\text{linf}[(\alpha_1, x_1); \ldots; (\alpha_k, x_k)]$. Also, variables x_i can have different names (like $var1$, $x4$, $y34$, etc.), and after the conversion into linf, all elements in the list will be sorted using some fixed ordering on the names of variables (usually, it is a lexicographic ordering). For efficiency, it is important to assume that the variables in the objective function $\mathbf{c}^T\mathbf{x}$ (i.e., variables for which $c_i \neq 0$) are the last one in the fixed ordering (we can always satisfy this assumption by renaming the variables).

First of all, we need to multiply each inequality by the corresponding value $0 \leq y_i$. It is a straightforward computation based on the following easy theorem

$$\vdash c * \text{linf}[(a_1, x_1); \ldots; (a_k, x_k)] = \text{linf}[(c * a_1, x_1); \ldots (c * a_k, x_k)]$$

Note that we can completely ignore inequalities for which $y_i = 0$ because they do not contribute to the sum which we want to compute.

The main step is computation of the sum of two linear functions. Suppose we have two linear functions $\text{linf}[(a, x_i); t_1]$ and $\text{linf}[(b, x_j); t_2]$ (t_1 and t_2 denote tails of the lists of pairs). Depending on the relation between x_i and x_j (i.e., we compare the names of variables), we need to consider three cases: $x_i \equiv x_j$ (the same variables), $x_i \prec x_j$ (in the fixed ordering), or $x_i \succ x_j$. In the first case, we apply the following theorem

$$\vdash \text{linf}[(a, x); t_1] + \text{linf}[(b, x); t_2] = (a + b) * x + (\text{linf}[t_1] + \text{linf}[t_2])$$

In the second case, we have the theorem

$$\vdash \text{linf}[(a, x_i); t_1] + \text{linf}[(b, x_j); t_2] = a * x_i + (\text{linf}[t_1] + \text{linf}[(b, x_j); t_2])$$

In the third case, the result is analogous to the second case. After applying one of these theorems, we recursively compute the expression in the parentheses and

$a+b$ (if necessary). Then we can apply the following simple result and finish the computation of the sum

$$\vdash a * x + \mathrm{linf}[t] = \mathrm{linf}[(a, x); t]$$

Moreover, if the variables in both summands are ordered, then the variables in the result will be ordered.

After adding all inequalities for constraints, we get the inequality $\mathbf{z}^T\mathbf{A}\mathbf{x} \leq \mathbf{z}^T\mathbf{b}$ where the left hand side is computed in terms of linf. Now we need to find the sum of this inequality and inequalities for boundaries (multiplied by the corresponding coefficients). We do not transform boundary inequalities into the linf representation. Each boundary inequality has one of two forms

$$\vdash -x_i \leq -l_i \text{ or } \vdash x_i \leq u_i$$

Again, we need to multiply these inequality by the corresponding element of $\mathbf{y} = (\mathbf{z}^T, \mathbf{v}^T, \mathbf{w}^T)^T$ and get

$$\vdash -v_i * x_i \leq v_i * -l_i \text{ or } \vdash w_i * x_i \leq w_i * u_i$$

If $v_i = 0$ or $w_i = 0$, then we can ignore the corresponding inequality. If for some x_i we have both boundary inequalities (lower and upper bounds) with non-zero coefficients, then find the sum of two such inequalities. After that, we get one inequality of the form $r_i x_i \leq d_i$ for each variable x_i.

Before finding the sum of the inequality $\mathbf{z}^T\mathbf{A}\mathbf{x} \leq \mathbf{z}^T\mathbf{b}$ and the boundary inequalities, we sort boundary inequalities using the same ordering we used for sorting variables in linf. We assumed that the variables in the objective function $\mathbf{c}^T\mathbf{x}$ are the last ones in our ordering. Let $c_i = 0$ for all $i \leq n_0$. Suppose that we want to find the sum of $r_1 x_1 \leq d_1$ and $\mathrm{linf}[(a_1, x_1); t] \leq s$. We know that $c_1 = 0$, so the first term $a_1 x_1$ in the linear function and $r_1 x_1$ must cancel each other, hence we have $a_1 = -r_1$. The sum can be found using the following result

$$\vdash a * x + \mathrm{linf}[(-a, x); t] = \mathrm{linf}[t]$$

So we can efficiently compute the sum of all boundary inequalities for $i = 1, \ldots, n_0$. For the last $n - n_0$ variables, we have non-vanishing terms and the sum can be found in the standard way. Practically, the number $n - n_0$ is small compared to n, so most of computations are done in the efficient way.

At last, we get the inequality

$$\vdash \mathrm{linf}[(c_{n_0+1}, x_{n_0+1}); \ldots; (c_n, x_n)] \leq M',$$

where $M' = \mathbf{y}^T\mathbf{b} \leq K$. It is left to prove that $M' \leq K$. This can be done using standard HOL Light procedures because we need to perform this operation only once for each linear program.

3 Optimization

We implemented our algorithm using HOL Light elementary inference rules as much as possible. We avoided powerful but time consuming operations (such as

tactics, rewrites, and derived rules) everywhere. Formal arithmetic operations play a very important role in our algorithm. The optimization of these operations is described below.

3.1 Integer Arithmetic

Formal arithmetic operations on integers are considerably faster than operations on rational (decimal) numbers. We want to perform all formal computations using integer numbers only. We have the following numerical values: entries of $\bar{\mathbf{A}}$, $\bar{\mathbf{b}}$, $\mathbf{c}$, $\mathbf{y}$, and the constant K. The input data $\bar{\mathbf{A}}$, $\bar{\mathbf{b}}$, $\mathbf{c}$, and K can be approximated by finite decimal numbers with at most p_1 decimal digits after the decimal point. The dual solution $\mathbf{y}$ is constructed in such a way that all its elements have at most p_2 decimal digits after the decimal point.

We modify the main step of the algorithm as follows. Compute

$$(10^{p_1+p_2}\mathbf{c}^T)\mathbf{x} = (10^{p_2}\mathbf{y}^T)\left(10^{p_1}\bar{\mathbf{A}}\right)\mathbf{x} \le (10^{p_2}\mathbf{y}^T)(10^{p_1}\bar{\mathbf{b}}) \le 10^{p_1+p_2}K.$$

It is clear, that the computations above can be done using integer numbers only. In the last step, we divide both sides by $10^{p_1+p_2}$ (using formal rational arithmetic only one time) and obtain the main inequality.

3.2 Faster Low Precision Arithmetic in HOL Light

We have obtained significant improvements in performance over the original implementations of basic arithmetic operations on natural numbers (operations on integers and rational numbers are derived from operations on natural numbers) by changing the internal representation of numerals. We are mostly interested in operations on relatively small numbers (5-20 decimal digits), so we do not consider improved arithmetic algorithms which usually work well on quite large numbers. Instead, we are using tables of pre-proved theorems which allow to get the computation results quickly for not very large numbers.

Numerals in HOL Light are constructed using three constants BIT0, BIT1, and 0 which are defined by

$$\mathrm{BIT0}(n) = n + n, \quad \mathrm{BIT1}(n) = n + n + 1.$$

Here, n is any natural number (it is expected that n is already in the correct form). With these constants, any numeral is represented by its binary expansion with the least significant bit first:

$$1 = \mathrm{BIT1}(0), \quad 2 = 1 + 1 = \mathrm{BIT0}(1) = \mathrm{BIT0}(\mathrm{BIT1}(0)), \text{ etc.}$$

Define new constants for constructing natural numbers. Instead of base 2, represent a number using an arbitrary base $b \ge 2$. When the base is fixed, we define constants

$$D_i(n) = bn + i.$$

In HOL Light, we write $D0$, $D1$, $D2$, etc. For example, if $b = 10$, then we can write $123 = D3(D2(D1(0)))$.

We implemented arithmetic operations on numbers represented by our constants in a straightforward way. As an example, consider how the addition operation is implemented for our numerals. First of all, we prove several theorems which show how to add two digits. If $i + j = k < b$, then we have

$$\vdash D_i(m) + D_j(n) = D_k(m + n)$$

If $i + j = k \geq b$, then

$$\vdash D_i(m) + D_j(n) = D_{k-b}(\mathrm{SUC}(m + n))$$

Here $\mathrm{SUC}(n) = n + 1$ is another arithmetic operation which is implemented for our numerals. Also we have two terminal cases $\vdash n + 0 = n$ and $\vdash 0 + n = n$.

We store all these theorems in a hash table. The names of the constants are used as key values: the theorem with the left hand side $D_1(m) + D_2(n)$ has the key value "D1D2". With all these theorems, the addition of two numbers is easy. Consider an example. Suppose $b = 10$ and we want to compute $14 + 7 = D4(D1(0)) + D7(0)$. First, we look at the least significant digits $D4$ and $D7$ and find the corresponding theorem $D4(m) + D7(n) = D1(\mathrm{SUC}(m + n))$. In our case, instantiate the variables by $m = D1(0)$ and $n = 0$. We obtain $D4(D1(0)) + D7(0) = D1(\mathrm{SUC}(D1(0) + 0))$. Recursively compute $D1(0) + 0 = D1(0)$ (it is the terminal case). Then we apply the procedure for computing $\mathrm{SUC}(n)$ and obtain $\mathrm{SUC}(D1(0)) = D2(0)$. Hence, the final result is obtained: $D4(D1(0)) + D7(0) = D1(D2(0))$.

The multiplication of two numbers is more complicated but also straightforward. The original procedure for multiplying two natural numbers in HOL Light is based on the Karatsuba algorithm [6]. This algorithm asymptotically faster than a naive approach but we found that in our case (numbers with 10-30 digits) the Karatsuba algorithm does not give a significant advantage and can even slow computations down on small numbers.

The subtraction and division are implemented in the same way as in HOL Light. Initially, the result of an operation is obtained using "informal" computer arithmetic, and then simple theorems along with formal addition and multiplication operations are used to prove that the computed result is indeed the right one.

4 Performance Tests

The results of performance tests for the improved multiplication and addition operations are given in Tables 1 and 2. These results are obtained by performing formal operations on 1000 pairs of randomly generated integer numbers.

Table 3 contains the performance results for our verification procedure of bounds of several Flyspeck linear programs. For each linear program, two test results are given. In one test, the native HOL Light formal arithmetic is used; in another test, the improved arithmetic with the fixed base 256 is utilized.

Note that most of the Flyspeck linear programs can be solved in less than 5 seconds. The linear programs in the table are selected to demonstrate a wide range of results.

Table 1. Performance results for 1000 multiplication operations

Size of operands	Native HOL Light mult.	Base 16 mult.	Base 256 mult.
5 decimal digits	2.220 s	0.428 s	0.148 s
10 decimal digits	7.216 s	1.292 s	0.376 s
15 decimal digits	16.081 s	3.880 s	1.316 s
20 decimal digits	59.160 s	6.092 s	2.256 s
25 decimal digits	85.081 s	10.645 s	3.592 s

Table 2. Performance results for 1000 addition operations

Size of operands	Native HOL Light add.	Base 16 add.	Base 256 add.
5 decimal digits	0.188 s	0.064 s	0.052 s
10 decimal digits	0.324 s	0.100 s	0.064 s
15 decimal digits	0.492 s	0.112 s	0.096 s
20 decimal digits	0.648 s	0.176 s	0.108 s
25 decimal digits	0.808 s	0.208 s	0.132 s

Table 3. Performance results for verification of linear program bounds

Linear program ID	# variables	# constraints	Native arith.	Base 256 arith.
18288526809	743	519	4.048 s	2.772 s
168941837467	750	591	5.096 s	3.196 s
25168582633	784	700	8.392 s	4.308 s
72274026085	824	773	7.656 s	5.120 s
28820130324	875	848	9.292 s	5.680 s
202732667936	912	875	9.045 s	5.816 s
156588677070	920	804	8.113 s	5.252 s
123040027899	1074	1002	11.549 s	6.664 s
110999880825	1114	1000	10.085 s	6.780 s

References

1. Gonthier, G.: Formal Proof—The Four-Color Theorem. Notices of the AMS 55(11), 1382–1393 (2008)
2. Hales, T.C.: The Flyspeck Project, http://code.google.com/p/flyspeck
3. Hales, T.C.: A proof of the Kepler conjecture. Annals of Mathematics. Second Series 162(3), 1065–1185 (2005)
4. Hales, T.C.: Linear Programs for the Kepler Conjecture (Extended Abstract). In: Fukuda, K., Hoeven, J.v.d., Joswig, M., Takayama, N. (eds.) ICMS 2010. LNCS, vol. 6327, pp. 149–151. Springer, Heidelberg (2010)

5. Harrison, J.: The HOL Light theorem prover, http://www.cl.cam.ac.uk/~jrh13/hol-light/
6. Karatsuba, A.A.: The Complexity of Computations. Proceedings of the Steklov Institute of Mathematics 211, 169–183 (1995)
7. Makhorin, A.O.: GNU Linear Programming Kit, http://www.gnu.org/software/glpk/
8. Obua, S.: Flyspeck II: The Basic Linear Programs (2008), http://code.google.com/p/flyspeck
9. A Modeling Language for Mathematical Programming, http://www.ampl.com/
10. The Caml Language, http://caml.inria.fr/

Large Formal Wikis: Issues and Solutions

Jesse Alama[1], Kasper Brink[2], Lionel Mamane, and Josef Urban[2,⋆]

[1] Center for Artificial Intelligence
New University of Lisbon
j.alama@fct.unl.pt, lionel@mamane.lu
[2] Institute for Computing and Information Sciences
Radboud University Nijmegen
josef.urban@gmail.com

Abstract. We present several steps towards large formal mathematical wikis. The Coq proof assistant together with the CoRN repository are added to the pool of systems handled by the general wiki system described in [10]. A smart re-verification scheme for the large formal libraries in the wiki is suggested for Mizar/MML and Coq/CoRN, based on recently developed precise tracking of mathematical dependencies. We propose to use features of state-of-the-art filesystems to allow real-time cloning and sandboxing of the entire libraries, allowing also to extend the wiki to a true multi-user collaborative area. A number of related issues are discussed.

1 Overview

This paper proposes several steps towards large formal mathematical wikis. In Section 3 we describe how the Coq proof assistant together with the CoRN repository are added to the pool of systems fully handled by the wiki architecture proposed in [10], i.e., allowing both web-based and version-control-based updates of the CoRN wiki, using smart (parallelized) verification over the whole CoRN library as a consistency guard. Because the task of large-scale library refactoring is still resource-intensive, an even smarter re-verification scheme for the large formal libraries is suggested for Mizar/MML and Coq/CoRN, based on precise tracking of mathematical dependencies that we started to develop recently for the Coq and Mizar proof assistants, see Section 4. We argue for the need of an architecture allowing easy sandboxing and thus easy cloning of the whole large libraries. This poses technical challenges in the real-time wiki setting, as cloning and re-verification of large formal libraries can be both a time and space

⋆ The first author was funded by the FCT project "Dialogical Foundations of Semantics" (DiFoS) in the ESF EuroCoRes programme LogICCC (FCT LogICCC/0001/2007). The third author was supported during part of the research presented here by the NWO project "Formal Interactive Mathematical Documents: Creation and Presentation"; during that time he was affiliated with the ICIS, Radboud University Nijmegen. The fourth author was supported by the NWO project "MathWiki a Web-based Collaborative Authoring Environment for Formal Proofs".

J.H. Davenport et al. (Eds.): Calculemus/MKM 2011, LNAI 6824, pp. 133–148, 2011.

consuming operation. An experimental solution based on the use of modern filesystems (Btrfs or ZFS in our case) is suggested in our setting in Section 5. Solving the problem of having many similar sandboxes and clones despite their large sizes allows us to use the wiki as a hosting platform for many collaborating users. We propose to use the gitolite system for this purpose, and explain the overall architecture in Section 6. As a corollary to the architecture based on powerful version control systems, we get distributed wiki synchronization almost for free. In section 7 we conduct an experiment synchronizing our wikis on servers in Nijmegen and in Edmonton. Finally we discuss a number of issues related to the project, and draw recommendations for existing proof assistants in Section 8.

2 Introduction: Developing Formal Math Wikis

This paper describes a third iteration in the MathWiki development.[1] An agile software development cycle typically includes several (many) loops of requirements analysis, prototyping, coding, and testing. A wiki for formal mathematics is an example of a strong need for the agile approach: It is a new kind of software taking ideas from wikis, source-code hosting systems, version control systems, interactive verification tools and specialized editors, and strong semantic-based code/proof assistants. Building of formal wikis seem to significantly interact with the development of proof assistants, and their mutual feedback influences the development of both. For example, a number of changes has already been done in the last year to the Mizar XML and HTML-ization code, and to the MML verification scripts, to accommodate the appearing wiki functionalities. See below for changes and recommendations to the related Coq mechanisms, and other possibly wiki-handled proof assistants. Also, see below in Section 4 for the new wiki functions that are allowed when precise dependency information about the formal libraries becomes available for a proof assistant.

The previous two iterations of our wiki development were necessarily exploratory; our work then focused on implementing the reasonably recognized cornerstone features of wikis. We used version control mechanisms suitable both for occasional users (using web interfaces) and for power users (working typically locally), and allowing also easy migration to future more advanced models based on the version-controlled repositories. We supplied HTML presentations of our content, enriched in various ways to make it suitable for formal mathematics (e.g., linking and otherwise improved presentation of definitions and theorems, explicit explanation of current goals of the verifier, etc.) One novel problem in the formal mathematical context was the need to enforce validity checks on the submitted content; for this, we developed a model of fast (parallelized) automated large-scale verification, done consistently for the largest formal library available.

[1] The first was an experimental embedding of the CoRN and MML repositories inside the ikiwiki (`http://ikiwiki.info/`) system, and the second iteration is described in our previous paper [10].

The previous implementations already provide valuable services to the proof assistant users, but we focused initially only on the Mizar proof assistant. While library-scale refactoring and proof checking is a very powerful feature of the formal wikis (differentiating them for example from code repositories), it is still too slow for large libraries to allow its unlimited use in anonymous setting. We have observed that users are often too shy to edit the main official wiki, as their actions will be visible to the whole world and influencing the rest of the users. A more structured/hierarchical/private way of developing, together with mechanisms for collaboration and propagation of changes from private experiments to main public branches are needed. Our limited implementation provided real-world feedback for the next steps described in this paper:

- We add Coq with CoRN to the pool of managed systems.
- We describe a smarter and faster verification modes for the wikis, that we started to implement within proof assistants exactly because of the feedback from previous wiki instances.
- We add a more fine-grained way to edit formal mathematical texts, making it easier to detect limited changes (and thus avoid expensive re-verification).
- We manage and control users and their rights, allowing the wiki to be exposed to the world in a structured way not limited to a trusted community of users.
- A mechanism in which the users get their own private space is proposed and tested, which turns out to be reasonably cheap thanks to usage of advanced filesystems and its crosslinking with the version control model.
- A high-level development model is suggested for the formal wiki, designed after a recently proposed model [6] for version-controlled software development. We extend that model by applying different correctness policies, which helps to resolve the tradeoffs between correctness, incrementality, and unified presentation discussed in [10].

One aim of our work is to try to improve the visibility and usability of formal mathematics. The field is sorely lacking an attractive, simple, discoverable way of working with its tools. The formal mathematics wiki we describe here is one project designed to tackle this problem.

3 The Generalized Formal Wiki Architecture, and Its Coq and CoRN Instance

One of the goals of initially developing a wiki for one system (Mizar) was to find out how much work is needed for a particular proof assistant so that a first-cut formal wiki could be produced. An advantage of that approach was that as Mizar developers we were capable to quickly develop the missing tools, and adjust the existing ones. Another advantage of focusing on Mizar initially was that the Mizar Mathematical Library (MML) is one of the largest formal mathematical libraries available, thus forcing us to deal early on with efficiency issues that go far beyond toy-system prototypes, and are seen in other formal libraries to a lesser extent.

The feasibility of the Mizar/MML wiki prototype suggested that our general architecture should be reasonably adaptable to any formal proof assistant possessing certain basic properties. The three important features of Mizar making the prototype feasible seem to be: batch-mode (preferably easily parallelizable) verification; fast dependency extraction (allowing some measure of intelligence in library re-compilation based on the changed dependencies); and availability of tools for generating HTML representations of formal texts. With suitable adaptation, then, any proof assistant with these properties can, in principle, be added to our pool of supported systems.

It turns out that the Coq system, and specifically the Coq Repository at Nijmegen (CoRN) formal library, satisfies these conditions quite well, allowing to largely re-use the architecture built for Mizar in a Coq/CoRN wiki[2].

3.1 HTML Presentation of Coq Content with Coqdoc

We found that the coqdoc tool, part of the standard Coq distribution, provides a reasonable option for enriched HTML presentation of Coq articles. With some additional work, it can be readily used for the wiki functionalities. Note that an additional layer (called Proviola) on top of coqdoc is being developed [8], with the goal of eventually providing better presentation and other features for interacting with Coq formalization in the web setting. As in the case of Mizar (and perhaps even more with nondeclarative proofs such as those of Coq), much implicit information becomes available only during proof processing, and such information is quite useful for the readers: For example, G. Gonthier, a Coq formalizer heading the Math Components project,[3] asserts that his advanced proofs are human-readable, however only in the special environment provided by the chosen Coq user interface. This obviously can be improved, both by providing better (declarative) proof styles for Coq (in the spirit of [5]), and by exporting the wealth of implicit proof information in an easily consumable form, e.g., similarly as Mizar does [9].

Unlike the Mizar HTML-ization tools (with possible exception of the MML Query tool [4]), the coqdoc tool provides some additional functionalities like automated creation of indexes and tables of contents, see for example Figure 1 for the CoRN wiki contents page. This can be used for additional useful presentation of the Coq wiki files, and is again a motivation (for Mizar and other proof assistants) to supply such tools for their wikis.

3.2 Batch-Mode Processing and Dependency Analysis with Coq

Coq allows both interactive and batch-mode verification (using the coqc tool), and also provides a special tool (coqdep) for discovering dependencies between Coq files, suitable for Makefile-based compilation and its parallelization. A difference of CoRN to MML is that the article structure is not flat in CoRN

[2] http://mws.cs.ru.nl/cwiki/

[3] ttp://www.msr-inria.inria.fr/Projects/math-components

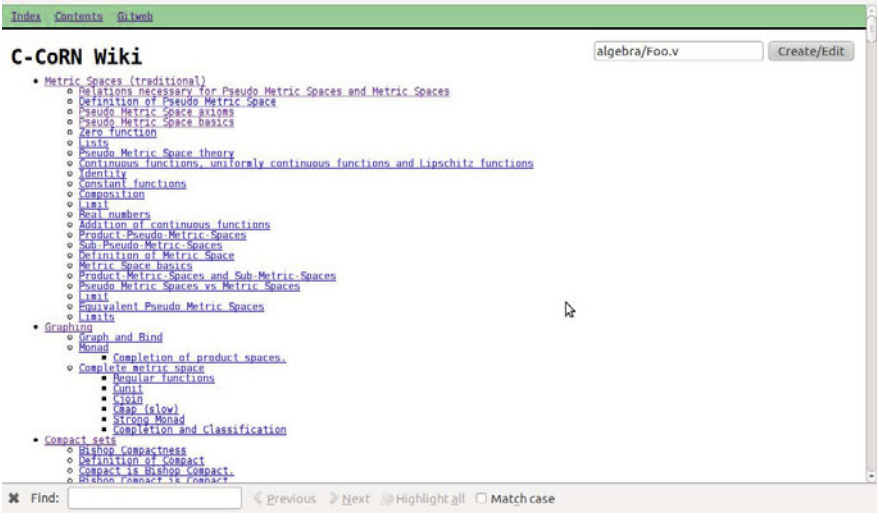


Fig. 1. CoRN wiki contents page

(in Mizar, all articles are just kept in one "mml" directory), and arbitrarily deep directory structure has to be allowed. This poses certain challenges when adding new files to CoRN, and taking care of their proper compilation and HTML presentation. The current solution is that the formal articles are really allowed to live in nested subdirectories, while the corresponding HTML live in just one (flat) directory (this is how the coqdoc documentation is traditionally produced), and the correspondence between the HTML and the original article (necessary for editing operations) is recovered by relying on the coqdoc names of the HTML files basically containing the directory (module) structure in them. This is a good example of a real-world library feature that complicates the life of formal wiki developers: It would be much easier to design a flat-structured wiki on the paper, however, if we want to cater for real users and existing libraries, imperfect solutions corresponding to the real world have to be used.

Interestingly, the structure of the dependencies in the CoRN repository differs significantly from the MML. MML can really benefit a lot from large-scale parallelization of the verification and HTML-ization, probably because it contains many different mathematical developments that are related only indirectly (e.g., by being based in set theory, using some basic facts about set-theoretic functions and relations, etc.). This is far from true for the CoRN library. Parallelization of the CoRN verification helps comparatively little, quite likely because the CoRN development is very focused. Thus, even though the CoRN library is significantly smaller than the MML (about a quarter of the size of the MML), the library re-verification times are not significantly different when verification is parallelized. This is a motivation for the work on finer dependencies described in Section 4.

3.3 New CoRN Development with SSReflect

A significant issue for wiki development turns out to be the new experimental version of CoRN, developed at Nijmegen based on the Math Components SSReflect library. This again demonstrates some of the real-world choices that we face as wiki developers. The first issue is binary incompatibility. The SSReflect (Math Components) project has introduced its own special version of the coqc binary, and standard coqc is no longer usable with it. Obviously, providing a common wiki for the Coq Standard Library and the Math Components project (even though both are officially Coq-based) is thus (strictly speaking) a fiction. One possible solution is that the compiled (.vo) files might still be compatible, thus allowing us to provide some clever recompilation mechanisms for the combined libraries. The situation is even worse with the developing version of CoRN, which relies (due to its advanced exploration of Coq type classes [7]) on both a special (fixed) version of the coqc binary, together with a special (fixed) version of the SSReflect library. This not only makes a joint wiki with the Coq Standard Library hard to implement, but it also prevents a joint wiki with the Math Components project (making changes to the SSReflect library, which has to be fixed for CoRN). To handle such real issues, the separate/private clones/branches of the wiki, used for developing certain features and for other experiments will have to be used. This is one of the motivations for our general proposal in Sections 5 and Section 6. It is noteworthy that older versions of CoRN also relied on their own Coq binary, including custom ML code. However, the features implemented by custom ML code were partly provided by newer versions of Coq, and partly reimplemented in Coq's LTac language. So there is a pattern there of new developments requiring custom Coq binaries which has to be taken into account when developing real-world wikis.

4 Using Fine-Grained Dependency Information for a Large Formal Wiki

In order to deal with the efficiency issues mentioned in previous sections, we have started to develop tools allowing much finer dependency tracking, and thus much finer and leaner recompilation modes, than is currently possible with Mizar and Coq. This work is reported in [1]. To summarize, we add a special dependency-tracking code to Coq, which can now track most of the mutual dependencies of Coq items (theorems, definitions, etc.), and extract the direct and transitive graph of dependencies between these items. Similarly, but using a different technique, we extract such fine dependencies from the Mizar formalizations. For Mizar this is done by advanced refactoring of the Mizar articles into one-item micro-articles, and computing their minimal dependencies by a brute-force minimization algorithm. The result of the algorithm again provides us for each item I with the precise information about which other Mizar items the item I depends on. This information is again compiled into graphs of direct and indirect dependencies. The Mizar wiki already allows viewing of fine theorem and scheme

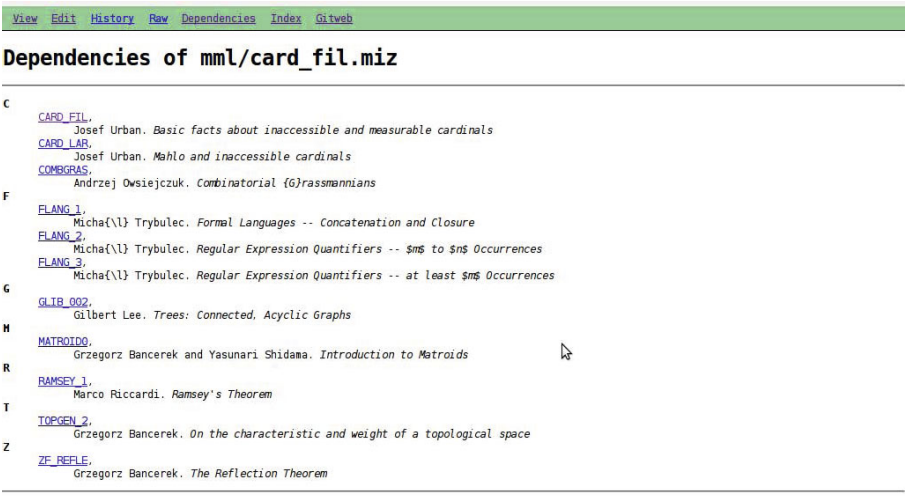


Fig. 2. Aggregated fine theorem and scheme dependencies for article CARD_LAR

dependencies aggregated for the articles, see Figure 2 for those of the CARD_LAR article.

4.1 Speeding Up (re)verification

It turns out that such fine dependencies have the potential to provide significant speedups for expensive library refactorings. The following Table 1 from [1] shows the dependency statistics and comparison for the CoRN and MML (first 100 articles) libraries. For example, the number of direct dependency edges computed by the fine-grained method in MML drops to 3% in comparison with the number of direct dependencies assumed by the traditional coarse file-based dependencies. This is obviously a great opportunity for the formal wiki providing very fast (and also much more parallalizable) verification and presentation services to the authors of formal mathematics.

4.2 Delimited Editing

The wiki now also exploits fine-grained dependency information, for the case of Mizar, by providing delimited text editing. The idea is to present the user with a way to edit parts of a formal mathematical text, rather than an entire article. This is a formal analog of the "Edit this section" button in Wikipedia. The task is to divide a text into its constituent pieces, and provide ways of editing only those pieces, leaving other parts intact. The practical advantage of such a feature is that we can be sure that edits to the text have been made only in a small part of the text that can have only a limited impact on other parts. When we know that an edit is made only to, say, the proof of a single theorem, then we do not need to check other theorem in the text; the text as a whole is correct just in

Table 1. Statistics of the item-based and file-based dependencies for CoRN and MML

	CoRN/item	CoRN/file	MML-100/item	MML-100/file
Items	9 462	9 462	9 553	9 553
Deps	175 407	2 214 396	704 513	21 082 287
TDeps	3 614 445	24 385 358	7 258 546	34 974 804
P(%)	8	54.5	15.9	76.7
ARL	382	2 577.2	759.8	3 661.1
MRL	12.5	1 183	155.5	2 377.5

Deps Number of dependency edges
TDeps Number of transitive dependency edges
P Probability that given two randomly chosen items, one depends (directly or indirectly) on the other, or vice-versa.
ARL Average number of items recompiled if one item is changed.
MRL Median number of items recompiled if one item is changed.

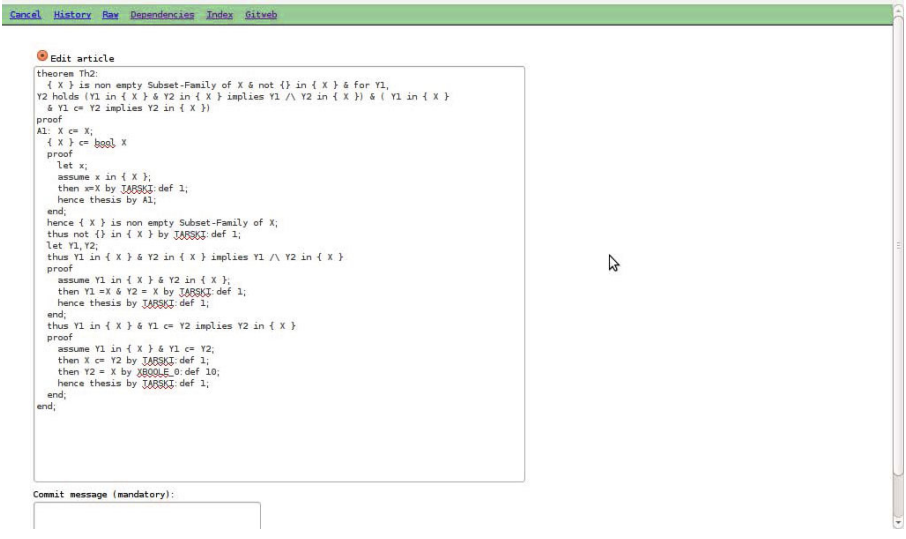


Fig. 3. Delimited editing of theorem `CARD_LAR:2`

case the new proof is correct. If the statement of a theorem itself is modified, it is sufficient to re-check only those other parts of the article that explicitly use or otherwise directly depend on this theorem. See Figure 3 for an example of delimited editing of theorem `CARD_LAR:2`.

5 Scaling Up

In Section 6 we propose a wiki architecture that caters for many users and many related developments, using the gitolite tool, and authentication policies

for repository clones and branches. As mentioned in Section 3, this seems to be a pressing real-world issue, necessary for the various collaborative aspects of formalization. Such a solution, however, forces us to deal with many versions of the repositories, which are typically very large. The Mizar HTML itself is several gigabytes in size, and in order to be able to quickly re-compile the formal developments, we also have to keep all intermediate compilation files around. In addition to that, our previous implementation needed the space for at least two versions of all these files, so that we could quickly provide a fresh sandbox (with all the intermediate files in it) for a recompilation of only the newly modified articles, and so that we were able to quickly return to a clean saved state if a re-compilation in the sandbox fails. Thus, the size of the Mizar wiki could reach almost 20 Gigabytes.

It is clear that with these sizes, it becomes impractical to provide a private clone or a feature clone for hundreds (or even dozens) of interested users. Fortunately, we can solve this by using the *copy-on-write* capabilities of modern filesystems: these mechanisms enable us to create time- and space-efficient copies of branches in the wiki, storing only the changes with respect to the original branch.

Currently, there are several copy-on-write filesystems under active development; a well-known example is the ZFS filesystem, which was first released by Sun Microsystems in 2005. Unfortunately, although ZFS is open-source, license incompatibilities prevent it from being distributed as part of the Linux kernel (which we use to host the MathWiki system). More recently, work has begun on a filesystem called Btrfs[4], which aims to bring many of the features of ZFS to Linux. Included in the mainline kernel in 2009, it is not yet as stable as traditional Linux filesystems, but its copy-on-write snapshotting is already usable for our purposes. The functionality provided by Btrfs can be combined with the architecture suggested in Section 6 to create a system that will scale to large numbers of users and branches, which is described below.

The git repositories themselves are typically quite small, as they are compressed, contain only the source files (not the intermediate and HTML files), and additionally git allows reference sharing. Thus the main problem are the working copies that need to be present on the server for browsing and fast recompilation. However, these copies will typically share a lot of content, because the users typically modify only a small part of the large libraries, and typically start with the same main branch.

Our solution is to implement the cloning of new user repositories using Btrfs snapshots. That is, we keep a working copy of the main repository in a separate Btrfs volume, and create a *snapshot* (a writeable clone) of this whenever a user clones the repository. Due to the copy-on-write nature of Btrfs, this operation is efficient in terms of time and space: creating a snapshot takes 0.03 seconds (on desktop-class hardware), and 6 KB of disk space, even for cloning very large (10G big) volumes as the one containing the Mizar wiki. Thus, we can now

[4] This stands for "B-tree filesystem".

provide space for a very large number of clones and versions, and do it practically instantaneously.

As the snapshot is modified, disk usage grows proportionally to the size of the changes. Changing a file's metadata (e.g., updating its last-modified-time, as required for our fast recompilation feature) costs 10 KB on average (this is a one-time cost, paid only when the user really makes the effort and does some acceptable changes). Modifying the content of a file increases disk usage by the amount of newly written data, plus a fixed overhead of about 12 KB. We have found that in order to maximize the amount of sharing between related snapshots, it is advisable to disable file-access-time updates on the filesystem.[5]

Each time a repository fails to compile, and needs to be restored, we can roll back to a previous state by discarding the latest snapshot. This is also a fast operation, typically taking less than a second, and saving us the necessity to maintain another 10G-large sandbox for possibly destructive operations, and peridically using (slower) file-based synchronization (`rsync`) with the main wiki.

The following Table 2 documents the scalability of Btrfs and its usability in our setting. It summarizes the following experiment: The main public wiki is populated with the whole Mizar library, which together with all the intermediate and HTML files takes about 10G of an (uncompressed) Btrfs subvolume. Then we emulate 10, 100, and 200 experimental wiki clones based on the main public wiki. Each of the clones starts as a snapshot of the main public wiki, to which a user decides to add his new development (Mizar article) depending on nontrivial part of the library (article `CARD_1` [3] was used). The article is then verified and HTML-ized, trigerring also library-wise update of various fine-dependency indexes and HTML indexes. This process is done by running full-scale `make` process on the whole library, requiring reading of modification times of tens of thousands of files in the newly created clone. Despite that, the whole process is reasonably fast and real-time, and scales well even with hundreds clones. The whole operation takes 6.9 seconds per clone on average for 10 clones, and 7.2 seconds on average when creating 200 clones in a series. The average growth in overall filesystem consumption (for the new article, its intermediate files, and updated indeces) is 5.22MB per clone when testing with 10 clones, and 5.26MB when testing with 200 clones. To summarize, the total cost of providing 200 personalized 10G-big clones with a newly verified article in them is only about 1GB of storage.

Table 2. Time and space data for 10, 100, and 200 clones with a new article verification

clones	time (s)	disk usage (MB)		
		data	metadata	total
10	6.9	4.71	0.51	5.22
100	7.0	4.71	0.55	5.26
200	7.2	4.71	0.55	5.26

[5] Using the "`noatime,nodiratime`" filesystem options.

6 Many Users, Many Branches

The current system now presents one version of CoRN and the MML to the entire community. To help make the site more attractive and useful, we would like the wiki to be a place where one can store one's work-in-progress; one would store one's own formal mathematical texts and have a mechanism for interacting with other users and their work. One could then track one's own progress online, and possibly follow other people's work as well. It would be akin to a GitHub for formal mathematics. In this section we describe the Git-based infrastructure for implementing multiple users.

The idea of extending a wiki such as ours from one anonymous user to a secure, multiuser one, maintaining security while preserving time and space efficiency, presents a fair number of technical challenges. One basic question: how do we extend our Git-based model? Would we store *one* repository for everyone, with different branches for each user, or do we give each user his own repository? How would one deal with ensuring that different users don't interfere with the work of other users? How do we deal with multiple people trying to access a repository (or repositories)? Note that this also leads to the problem of storing many different (but only slightly different) copies of large formal corpora solved in the previous section by using advanced filesystem.

For managing multiple users we opted for a solution based on the gitolite system.[6] gitolite adds a layer to Git that provides for multiple users to access a pool of repositories, guarded by SSH keys. With gitolite one can even set up fine-grained control over particular branches of repositories. One can specify that certain repositories (or a particular branch) is unavailable to a user (or group of users), readable but not writable, or read-writable. gitolite makes transparent use of the SSH infrastructure; once a user has provided RSA public key to us (the registration page is shown in Figure 4), he is able to carry out these operations via the web page or through the traditional command-line interface to Git.

In addition to supporting multiple users, we also want to permit multiple branches per user. The following Git branching policy described by V. Driessen [6] provides a handful of categories of branches:

> We consider origin/master to be the main branch where the source code of HEAD always reflects a production-ready state. We consider origin/develop to be the main branch where the source code of HEAD always reflects a state with the latest delivered development changes for the next release. Some would call this the "integration branch". This is where any automatic nightly builds are built from.

In addition to *main* and *developer* branches, we intend to support other kinds of branches: *feature* (for work on a particular new feature), *release* (for official releases of the formal mathematical texts), and *hotfix* (fixes for critical bugs).

A gitolite access implementing a Driessen-style model can be seen in Figure 5.

[6] https://github.com/sitaramc/gitolite/wiki/

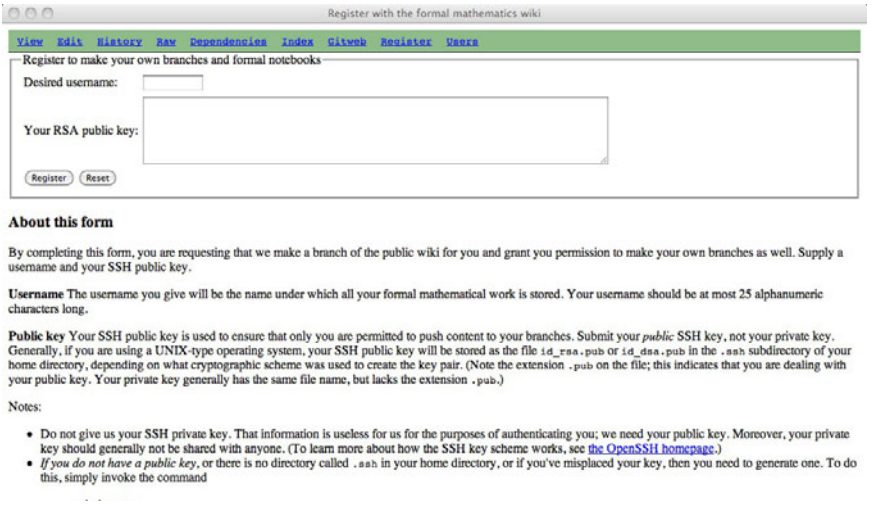


Fig. 4. Registration page at our wiki

```
@all = @superusers @maintainers @developers @users @anonymous

repo    main
        RW+     =   @superusers @maintainers
        R       =   @developers @users @anonymous

repo    devel
        RW+     =   @superusers @maintainers @developers
        R       =   @users @anonymous

repo    feature/[a-zA-Z0-9].*
        C       =   @superusers @maintainers @developers
        RW+     =   @superusers @maintainers @developers
        R       =   @users @anonymous

repo    (release|hotfix)/[a-zA-Z0-9].*
        C       =   @superusers @maintainers
        RW+     =   @superusers @maintainers
        R       =   @developers @users @anonymous

repo   user/CREATOR/[a-zA-Z0-9].*
       C       =   @superusers @maintainers @developers @users
       RW+     =   CREATOR
       R       =   @all
```

Fig. 5. A gitolite policy for different kinds of wiki users

The intention of this policy is to divide users into certain classes and permit certain kinds of operations (creating a branch, reading it, reading-and-writing to it). The user classes have the following meaning:

- `admin`: can do anything, has root access to the server
- `superuser`: can do arbitrary operations on the wikis taking arbitrary times, can update binaries, etc
- `maintainer`: can update the main stable wiki, start/close the release and hotfix branches
- `developer`: can update the develop clone, start/close feature branches,
- `user`: limited to his userspace, and inexpensive operations
- `anonymous`: limited to the anonymous user space

The name of the repository is now also an argument to a Git pre-commit or pre-receive hook, which applies a particular verification policy to the repository. For the *main* and *develop* repositories the policy should require full verifiability, while other branches should not have to, so that these function more like work-in-progress notebooks. (Such branches present an interesting problem of displaying, in a helpful way, possibly incorrect formal mathematical texts). gitolite also provides a locking mechanism for addressing the problem of concurrent reads and writes.

With the registration form, the wiki users can now submit their RSA public keys to the wiki system. Doing so adds them to gitolite's user space, so that they can create new (frontend) Git repositories (e.g., by cloning some already existing repository). Doing so triggers the creation of a corresponding backend repository (gitolite manages directly the frontends, while the backend is managed indirectly by us via Git hooks and CGI). The backend repositories contain the full wiki populated with the necessary intermediate files needed for fast re-comopilation, and obviously also with the final HTML representation of the contents, exactly as we did in the previous one-user, one-repository version of MathWiki. The backends themselves live in a filesystem setup described in Section 5 that reuses space using filesystem techniques as copy-on-write. The result is quite a scalable platform, allowing many users, many (related) developments, different verification and authorization policies via gitolite and git hooks, and attempting to provide as fast verification and HTML-ization services as possible for a given proof assistant and library.

Note however that tasks such as re-verifying a whole large library from scratch will always be expensive and this should be reflected to the users. Apart from the many efficiency solutions mentioned so far, we are also experimenting with the problem of queuing pending wiki operations. We should allow them to have various superficial fast modes of verification.[7] Users could have their own queues of jobs, and would be allowed to cancel them, if they see that some other task would invalidate the need to do the other ones. However, when committing to

[7] For Mizar, one could run only the exporter, or also the analyzer, or the full verifier. These "compiler-like" stages actually do not have to be repeated in Mizar once they were run.

the *devel* or *main* branches, as mentioned, full verification should always be required.

7 Multiple Wiki Servers and Their Synchronization

Mirroring is a common internet synchronization procedure used for a number of reasons. Mirroring increases availability by decreasing network latency in multiple geographical locations. Mirroring also helps to balance network loads and supports backup of content. An internet mirror is *live* when it is changed immediately after its origin changes. With custom wiki software, such as MediaWiki[8] (the wiki engine behind Wikipedia), there can typically be just one central repository to which updates are made. This is no longer such a limitation with a wiki such as our, which is built on top of a distributed version control system.

In case of the Mizar part of our wiki, the practical motivation for mirroring already exists: There are currently three reasonably powerful servers (in Nijmegen, Edmonton, and Bialystok) where the wiki can be installed and provide all its services. Given that re-verification of the whole formal (e.g., Mizar) library is still a costly operation, distributing the work between these servers can be quite useful. An obvious concern is then however the desynchronization of the developments.

This turns out to be easy to solve using the synchronization mechanism of a distributed version control system like Git. Git already comes with its own options for mirroring the changes in other repositories, which can be easily triggered using some of its hooks (in Git terminology, we are using the post-update hook on bare repositories). Because our wiki is "just" a Git repository (with all other functionalities implemented as appropriate hooks) that allows pushing into it as any other Git repository, it turns out that this mirroring functionality is immediately usable for live synchronization of our wikis. The process (for example, for two wikis) works as follows:

- The wikis are initialized over the same Git repository.
- A post-update hook is added to the frontend (bare) Git repository of each of the wikis, making a mirroring push (pushing of all new references) to the mirroring wiki's frontend repository.
- Upon a successful commit/push to any of the wiki servers, the pushed server thus automatically updates also the mirroring wiki, triggering its verification and HTML-ization functions, exactly in the same way as a normal push to the wiki triggers these wiki-updating functions.

Note that this is easy with distributed version control systems such as Git, precisely because there is no concept of a central repository, so that all repositories are equal to each other and implement the same functionality. It is easy also because from the very beginning, our wiki was designed to allow arbitrary remote pushes, not just standard wiki-like changes coming from web editing.

This mechanism also allows us to have finer mirroring policies. For example, a realistic scenario is that each of the wiki servers by default mirrors only changes

[8] http://www.mediawiki.org/wiki/MediaWiki

to the main public wiki branches/clones, and the private user branches are kept non-mirrored. This means that the potentially costly verification operation is not duplicated on the mirror(s) for local developments, and is done only when an important public change is made.

8 Conclusion and Further Issues

We have outlined a number of steps for building on our first version of a formal mathematics wiki. Our aims naturally require us to make use of several disparate technologies, including cutting-edge ones such as smart filesystems that can cope with very large scale datasets.

The ultimate aim of making formal mathematics more attractive and manageable to the everyday mathematician remains. Extending our idea of "research notebooks", we would eventually like to equip our wiki with an editor with which one's mathematical work could be carried out entirely on the web. Collaborative tools such as etherpad[9] are a natural target as well. Hooks into attractive, useful presentations of formal proofs such as Mamane's tmEgg and Tankink's Proviola [8] systems can help, and merging in powerful automation tools such as the MizAR system [11] is another obvious next step.

At the moment, our wiki supports only Mizar and Coq. These are but two of the actively used systems for formalized mathematics; adding Isabelle and possibly HOL light are now within reach thanks to our experience with Mizar and Coq. Concerning Coq, we would like to take advantage of the ongoing Math Components project.

Finally, we note that mappings between formal mathematics and the vast world of "informal" mathematics remains rather weak. Indeed, even links between formal repositories is rather underdeveloped. Linking formal mathematical texts to some informal counterparts, such as to Wikipedia, PlanetMath[10], Wolfram MathWorld[11], remains to be carried out. For Mizar, this has been achieved to some extent (providing Wikipedia-based mapping for about two hundred MML objects), but much remains to be done. It seems especially attractive, in the context of our wiki work, to build a well-connected corner of the World Wide Web linking formal and informal mathematics.

References

1. Alama, J., Mamane, L., Urban, J.: Dependencies in formal mathematics (preprint) (submitted)
2. Autexier, S., Calmet, J., Delahaye, D., Ion, P.D.F., Rideau, L., Rioboo, R., Sexton, A.P. (eds.): AISC 2010. LNCS, vol. 6167. Springer, Heidelberg (2010)
3. Bancerek, G.: Cardinal numbers. Formalized Mathematics 1(2), 377–382 (1990)

[9] http://etherpad.org/
[10] http://planetmath.org/
[11] http://mathworld.wolfram.com

4. Bancerek, G.: Information retrieval and rendering with MML Query. In: Borwein, J.M., Farmer, W.M. (eds.) MKM 2006. LNCS (LNAI), vol. 4108, pp. 266–279. Springer, Heidelberg (2006)
5. Corbineau, P.: A declarative language for the Coq proof assistant. In: Miculan, M., Scagnetto, I., Honsell, F. (eds.) TYPES 2007. LNCS, vol. 4941, pp. 69–84. Springer, Heidelberg (2008)
6. Driessen, V.: A successful Git branching model, http://nvie.com/posts/a-successful-git-branching-model/
7. Spitters, B., van der Weegen, E.: Developing the algebraic hierarchy with type classes in Coq. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 490–493. Springer, Heidelberg (2010)
8. Tankink, C., Geuvers, H., McKinna, J., Wiedijk, F.: Proviola: A tool for proof re-animation. In: Autexier, et al. (eds.) [2], pp. 440–454
9. Urban, J.: XML-izing mizar: Making semantic processing and presentation of mml easy. In: Kohlhase, M. (ed.) MKM 2005. LNCS (LNAI), vol. 3863, pp. 346–360. Springer, Heidelberg (2006)
10. Urban, J., Alama, J., Rudnicki, P., Geuvers, H.: A wiki for Mizar: Motivation, considerations, and initial prototype. In: Autexier, et al. (eds.) [2], pp. 455–469
11. Urban, J., Sutcliffe, G.: Automated reasoning and presentation support for formalizing mathematics in Mizar. In: Autexier, et al. (eds.) [2], pp. 132–146

Licensing the Mizar Mathematical Library

Jesse Alama[1], Michael Kohlhase[2], Lionel Mamane, Adam Naumowicz[3], Piotr Rudnicki[4], and Josef Urban[5,⋆]

[1] Center for Artificial Intelligence, New University of Lisbon
j.alama@fct.unl.pt
[2] Computer Science, Jacobs University
m.kohlhase@jacobs-university.de, lionel@mamane.lu
[3] Institute of Computer Science, University of Bialystok
adamn@math.uwb.edu.pl
[4] Department of Computing Science, University of Alberta
piotr@cs.ualberta.ca
[5] Institute for Computing and Information Sciences; Radboud University Nijmegen
josef.urban@gmail.com

Abstract. The Mizar Mathematical Library (MML) is a large corpus of formalised mathematical knowledge. It has been constructed over the course of many years by a large number of authors and maintainers. Yet the legal status of these efforts of the Mizar community has never been clarified. In 2010, after many years of loose deliberations, the community decided to investigate the issue of licensing the content of the MML, thereby clarifying and crystallizing the status of the texts, the text's authors, and the library's long-term maintainers. The community has settled on a copyright and license policy that suits the peculiar features of Mizar and its community. In this paper we discuss the copyright and license solutions. We offer our experience in the hopes that the communities of other libraries of formalised mathematical knowledge might take up the legal and scientific problems that we addressed for Mizar.

Keywords: free culture, open data, free licensing, formal mathematics, mizar.

1 Introduction – Formal Mathematics and Its Roots

The dream of formal thinking and formal mathematics (and its giant offspring: computer science) has a long and interesting history that we can hardly go into

⋆ The first author was funded by the FCT project "Dialogical Foundations of Semantics" (DiFoS) in the ESF EuroCoRes programme LogICCC (FCT LogICCC/0001/2007). The sixth author was supported by the NWO project "MathWiki a Web-based Collaborative Authoring Environment for Formal Proofs". The authors have worked together as members of the SUM Licensing Committee. Special thanks for advice to: Stephan Schulz, David Wheeler, Michael Spiegel, Petr Pudlak, Bob Boyer, Adam Pease, Timo Ewalds, Adam Grabowski, Czeslaw Bylinski, and Andrzej Trybulec.

J.H. Davenport et al. (Eds.): Calculemus/MKM 2011, LNAI 6824, pp. 149–163, 2011.

in this paper. Briefly, formal mathematics started to be produced in *computer understandable* encoding in late 1960s. The first significant text of formal mathematics was van Benthem's encoding of Landau's *Grundlagen der Analysis* [11] in AUTOMATH [1]. Since then, large bodies of formal mathematics have been created within the fields of Interactive and Automated Theorem Proving (ITP, ATP). (See [25] for an extensive overview of the systems, formal languages, and their libraries.)

As these libraries grow and their contents get re-used in new, possibly unimagined and unintended contexts, their legal status needs to be clarified and formalised. In this paper we discuss how this problem was tackled in the case of the Mizar Mathematical Library. In Section 2 we discuss formal mathematical libraries in general and the target library of interest for us, the Mizar Mathematical Library (MML [14]), and the problem of classifying its content as code vs. text. We discuss in Section 3 how some basic licensing problems were tackled by other formal mathematics projects. In Section 4 we survey the main issues we faced, and our (sometimes incomplete) solutions to them. We offer further problems and future work in Section 5. Our final copyright/licensing recommendation is accessible online[1], and the Mizar copyright assignment and licensing policy are now parts of the Mizar distribution[2].

2 What Is a Formal Math Library?

A library of formal mathematics is a collection of "articles" that contain formalised counterparts of everyday informal mathematics. Our interest here is on the Mizar Mathematical Library (MML). Here we discuss some of the historical background of this library and the problems it poses for the license designer.

2.1 Historical Background of the MML

The year 1989 marks the start of a systematic collection of inter-referenced Mizar articles. The first three articles were included into a Mizar data base on January 1 — this is the official date of starting the Mizar Mathematical Library — MML, although this name appeared later.

The copyright for the MML has been owned by the Association of Mizar Users (SUM, in Polish Stowarzyszenie Użytkowników Mizara) anchored in Białystok. SUM is a registered Polish association whose statute[3] states that the SUM's aim is popularizing, propagating and promoting the Mizar language. The copyright assignment has been required from the authors (typically SUM members) by

[1] https://github.com/JUrban/MMLLicense/raw/master/RECOMMENDATION

[2] http://mizar.org/version/current/doc/Mizar_FLA.pdf
http://mizar.org/version/current/doc/FAQ
http://mizar.org/version/current/doc/COPYING.interpretation
http://mizar.org/version/current/doc/COPYING.GPL
http://mizar.org/version/current/doc/COPYING.CC-BY-SA

[3] http://mizar.org/sum/statute.new.html

SUM when submitting articles to MML. This was likely related to the early decision to build a large re-usable library of formal mathematics, which would presumably be re-factored many times by the core MML developers. Another reason speaking for the copyright centralization was the fact that the potential uses of such a library were initially also quite unclear: note that MML was started so early that it precedes the World Wide Web, Linux, Wikipedia, arXiv, and the massive development of free software and free web resources like Wikipedia and arXiv in the last two decades, and the related development of free licenses.

While the MML copyright owner was always clearly SUM, it was nowhere stated what was covered by the copyright and there was no policy for licensing the use of MML content should someone request a permission for use other than contributing to the library. For example, Josef Urban was not sure whether it was legal to include his translation [19] of parts of the MML into the TPTP library, which is used also for testing/benchmarking of ATPs — a process with some potential for commercial use. As discussed in the next section, the ATP translation also makes MML more executable in certain sense, in a similar way as for example Prolog program can be viewed as a database and as a program simultaneously. Such potential uses add to the need for a clear licensing policy.

2.2 Two Aspects of Formal Mathematics: Code and Text

Formal mathematical libraries present a number of problems for the license designer. One crucial question for deciding upon a license for the Mizar Mathematical Library: Is a Mizar article more like a journal article, or is it more like a piece of computer code, or is it both? And could existing law be suboptimal in treating code differently from mathematical texts?

Although we are focused on Mizar, we note in passing that other interactive proof assistants, whose communities might want to take up the problem of licensing their content, face this issue in different ways. Definitions and proofs in Coq, for instance, have originally been rather more like computer programs (at least, prima facie) than Mizar texts[4].

We also need to address the issue of what can be done with formal mathematical texts. There is considerable interest in extracting algorithms from proofs of universal-existential theorems. What is the status of such extracted or derived products? Mizar does not, on its face, permit such a straightforward, immediate extraction of algorithms from proofs. There are however several mechanisms which bring Mizar very close to executable code:

1. Formal mathematical formulas (more precisely: clauses) immediately give rise to very real computation in the Prolog language. For example, the Prolog algorithm for reversing lists:

[4] This is however also changing: Coq has become capable of handling mainstream (not just constructive) mathematics recently, has gained a declarative proof mode and one of the aspirations of the Math Components project is to make the Coq presentations accessible to mathematicians.

```
reverse_of([],[]).
reverse_of([H|T],Result):-
   reverse_of(T,RevT),
   append(RevT,[H],Result).
```

is just two mathematical clauses (formulas):

$$rev\_of([],[]) \wedge$$
$$\forall H, T, RevT, Result : rev\_of(T, RevT) \wedge append(RevT, list(H,[]), Result)$$
$$\rightarrow rev\_of(list(H,T), Result)$$

The only difference between a Prolog program and a set of mathematical clauses is that the order of clauses matters in the Prolog program.

2. In systems for automated theorem proving (ATPs), even this ordering difference typically no longer exists. Many ATPs would thus be really able to "compute" the reverse of a particular list, just with the two formulas above, given to them in arbitrary order. The MPTP system [20] translates all Mizar formulas into a format usable by ATPs, and thus allows such computations to be made.
3. Mizar itself actually contains pieces of more procedural semantics, e.g. the "registrations" mechanisms (a kind of limited Prolog mechanism). These mechanisms add restricted Prolog-like directed search to otherwise less directed general Mizar proof search, in a similar spirit as the Prolog language adds a particular restrictions to the order in which (Horn) clauses are traversed, as opposed to ATPs that work with arbitrary clauses and regardless of ordering. Input to these mechanisms are again Mizar formulas in special (Horn-like) from.
4. In principle, one *could* extract a mangled form of constructive content from the classical content of the Mizar Mathematical Library by applying, say, the Gödel-Gentzen translation from classical to intuitionistic logic. After that, the Curry-Howard isomorphism between proofs and programs could again be used to give procedural meaning to Mizar proofs (not formulas as in the above cases).

In short, mathematics lends itself to executable interpretation not just via the Curry-Howard proofs-as-programs interpretation of constructive mathematics (reasonably well-known in the formal community), and extensions of it to classical mathematics, but also by implicit formulas-as-programs interpretations, developed by the Prolog and ATP communities. It is a well known fact that efficient Prolog computation is just a restriction of ATP proof search, and ATP proof search can be used as (typically less efficient than Prolog) computation too. These mechanisms are to a certain extent present already in Mizar, and are fully available using a number of systems via the MPTP translation.

The formal/informal distinction cannot be straightforwardly based the ability for a machine to extract content/meaning. For example, Wikipedia is today used for data and algorithm extraction, used in advanced algorithms by, for example, IBM Watson system [21]. With sufficiently advanced extraction algorithms

(which we are clearly approaching), many "documents" can become "programs". (For an interactive demonstration, see [10].)

Various other factors make contributions to the Mizar Mathematical Library like computer code. An entry in Wikipedia can stand alone as a sensible contribution. Mizar articles, however, do not stand alone (in general), because it cannot be verified — or even parsed — in the absence of the background provided by other articles. With some background knowledge in mathematics, some human-understandable meaning can be extracted from Mizar texts [15]:

```
reserve n,m for Element of NAT;
reserve g for Element of COMPLEX;
reserve s for Complex_Sequence;
definition
 let s;
 attr s is convergent means
 ex g st for p be Real st 0 < p
  ex n st for m st n <= m holds |.s.m-g.| < p;
end;
```

is evidently the conventional $\exists\forall\exists\forall$-definition of a convergent sequence of complex numbers. However, the exact meaning of this text can be specified only with reference to the environment in which this text is evaluated. The environment provides some type information, such as that `0` is a real number, `<` is a relation among real numbers, the curious-looking `|.s.m-g.|` is a real number (it is the absolute value of the difference of the nth term s_n and g), etc.

Thus, libraries of formal mathematics are akin to libraries of software. Code that calls a library function cannot function without the library. Similarly, a formal mathematical article is not "formal" as it cannot be understood in the absence of the other formal articles it imports. The background formal library is used as a declarative and procedural knowledge to derive (not just "verify") the contents of a new formal mathematical article.

2.3 Code and Text Licenses

Because of the dual nature of the formal mathematical texts — they are both human-readable (particularly when written in Mizar) and machine-processable — it is possible that we are dealing with a new kind of object. The licensing situation in the world of free works[5] within the two categories (executable software code on the one hand, and documents for human consumption on the other hand) has clear "winners". On the code side, statistics [16,9,24] based on web scraping of contents of large free/open source software repositories (such as SourceForge or freshmeat) show that slightly more than a half of FLOSS[6] code is under a variant

[5] "Free" refers not (only) to zero price (Latin gratis), but to freedom (Latin liber; however, the consecrated vernacular expression is "libre", so we will use "libre"). That is, a work that anybody is free to use, share and improve.

[6] FLOSS for *free/libre/open-source software*.

of the GNU General Public License: Data from the FLOSSmole project [4] as of March 2011 shows that out of a total of 43 470 FLOSS projects tracked by freshmeat, 24366 are licensed under a version of the GNU GPL[7]. Of the projects hosted on SourceForge, 110 412 use a variant of the GPL, out of 174 227 that use a license approved by the Open Source Initiative.

On the document side, although we were not able to find comparative usage statistics, in our experience Creative Commons accounts for most of the mind-share, although domain-specific licenses have fair success within their domain. See [6,7].

Be it only for this reason, it seemed prudent to us to allow for an eventual future relicensing, and for this to keep central copyright ownership of the MML. But also, the practice of licensing of free/libre electronic documents is rather younger, and less mature, than free software licensing, thereby increasing the "risk" that relicensing may be necessary in the future. For example, Wikipedia migrated from a GNU Free Documentation license to a Creative Commons BY-SA license as recently as in 2009, because a majority of other wikis had, by and large, settled for a Creative Commons license, and Wikipedia wished to make interchange of content between Wikipedia and other wikis legally possible (and easy).

An additional uncertainty arises from the fact that MML articles can be seen both as documents and as executable code; possibly difficulties could arise at some point from this dual nature. For example, someone wanting to make a use of the MML that sits squarely neither on the one side nor the other, but makes use of the duality in some way. Possibly this use would neither be clearly allowed by a license meant for executable code, nor clearly allowed by a license meant for documents. It is thus prudent to have a central authority that can authorise such uses on a case-by-case basis as they arise, or revise the license of the MML once the issues are better understood.

2.4 Patents

Restrictions arising from patents are potentially just as lethal as copyright restrictions for keeping a work free to use and enhance by anybody for any purpose. The expected content of the MML is however more of an abstract nature than of a technical nature. However, first, (theoretically) only technical ideas are subject to patents, notwithstanding the situation concerning software patents. Second, the kind of things that the MML is now typically used for would not infringe on a patent, even if the idea expressed in an MML article would be covered by a patent: a patent do not forbid the activity proper of studying or enhancing upon the covered idea; for example, the RSA or IDEA algorithms being patented does not forbid proving (formally or in paper mathematics) properties of these algorithms, nor does it forbid teaching the algorithm publicly. On the contrary, the usual justification for the modern patent system is to encourage inventors to make descriptions of their invention public, rather than keeping them as trade

[7] This count includes various versions of the GNU GPL, as well as the combination of the GNU GPL with special linking exceptions; it does not include significantly more liberal licenses, such as the GNU Lesser/Library GPL.

secrets, so that such activities can take place, and eventually lead to a larger and better exploitation of the idea by society as a whole. The activity forbidden by a patent is manufacturing, selling or using an *implementation* of the idea, a machine based on the idea. Eventual concrete advances in making a specification of an algorithm in Mizar executable could create interesting legal questions, but we are not yet at this point[8] although some preliminary experiments[9] were quite encouraging. Third, in the context of free software we take our inspiration from, how to handle patents is — by far — not as consensual as copyright licensing. Free documents, our other inspiration, usually don't have to deal with patent issues. For the combination of these three reasons, we did not address any patent issue in our initial licensing recommendation to SUM.

However it is worth noting that in GPL version 3 the FSF has started to address the issue of software patents more concretely, inserting following into the GPL v3 preamble:

> Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

Section 11 of the GPL v3 text itself contains a blanket patent license from every contributor in addition to the usual copyright license. In other words, GPL version 3 has the share-alike (transitive) aspect not only with respect to copyright, but also with respect to patents. However our copyright assignment setup means that a contributor not redistributing his MML modifications himself never agrees to the GPL, and thus evades its patent license provisions. Additionally as the Creative Commons license does not address patents, our dual-license model allows redistributors of modified versions to avoid giving a public patent license by choosing CC-BY-SA. Addressing patent issues (if relevant at all) is thus possible future work for us, probably subject to some discussions with the authors of the FSFE Fiduciary License Agreement (copyright assignment contract).

3 Related Licensing Models

For detailed overview of formal systems' licensing, see David Wheeler's enumeration [23]. The tendency in academic institutions over the last decade seems

[8] A similar question of where the boundary between "implementation" and "human speech" lies arose in the last years of the second millennium, concerning the CSS algorithm used to scramble the contents of Video DVDs, although in this case the problem did not originate from a patent, but from a then-new copyright meta-protection law, the USA Digital Millennium Copyright Act. See `http://www.cs.cmu.edu/~dst/DeCSS/Gallery/` for examples.

[9] Michal Michaels (`http://mmitech.net/michael/cv`) added computational contents to Mizar schemes and was extracting Lisp code for binary arithmetics in his MSc thesis, University of Alberta 1996.

to go from closed/non-free/non-commercial/unclear licensing terms on formal systems, towards open/free/clear ones. Two examples are the SPASS theorem prover from MPI Saarbrücken, and the PVS verification system from SRI. SPASS went from a custom license allowing only non-commercial use (SPASS 1.0) to GPL2 (SPASS 2.0) to FreeBSD license (SPASS 3.5). PVS has switched from a former commercial license to GPL as of December 2006[10]. One of the early rules (since 1997) of the CADE ATP System Competition (CASC) has been that "Winners are expected to provide public access to their system's source code",[11] and that the systems' sources are after the competition regularly published by the CASC organisers. Note that the situation is quite different in the world of more applied formal tools like SAT and SMT solvers. For example, neither the Z3 (Microsoft Research) nor the Yices (SRI) SMT solvers are FLOSS.

However, our particular interest are not the formal systems per se, but rather the formal libraries associated with them. As discussed in Section 2, this distinction — between the systems and the mathematics formalised inside them — might be (im)possible to various extents. For example, the HOL (Light) formalizations (and thus the large mathematical Flyspeck project formalizing the proof of the Kepler conjecture) are written directly in the ML (OCaml) programming language. This is probably best captured as "proof programming" (the ML code) over "mathematical terms" (specially parsed parts of the ML code). Obviously, arbitrary programs (ML functions) thus are part of the "procedural proofs" written in HOL (Light)[12]. On the other hand, in Mizar, the distinction between the system's code (written in Pascal) and the formalization code (written in Mizar) is very clear: no Pascal programming is allowed inside the declarative mathematical proofs.

3.1 Licensing Models of Formal Libraries

All major formal libraries have so far used code licenses, while informal ones like arXiv, PlanetMath and Wikipedia use document licenses. The GNU Free Documentation License (GFDL) has been previously used by Wikipedia and PlanetMath, however, as Stephan Schulz (a Wikipedia administrator) noted:

> The GFDL certainly is a reasonable choice. However, it has some warts, and large collaborative projects (in particular Wikipedia) have been moving (with support from the Free Software Foundation) to the Creative Commons CC-BY-SA license. The CC-BY-SA license allows

[10] http://pvs.csl.sri.com/mail-archive/pvs-announce/msg00007.html

[11] http://www.cs.miami.edu/~tptp/CASC/14/Call.html#Conditions

[12] One might of course argue that the arbitrary ML functions in HOL (Light) serve not as "parts of proofs" but rather as "proof generators", i.e., that the "real proof" is just the low-level HOL proof object (checked by HOL's LCF-like microkernel), which the user typically never sees. This is however a bit like saying that the Lisp (or C) macro language is not really a part of Lisp (or C) programming, or even that Lisp (or C) is just a "program generator", and the "real program" is just the compiled machine code.

Table 1. Overview of licenses of selected (formal) mathematical libraries

Library (System)	License
Coq Standard Library (Coq)	LGPL
Coq Repository at Nijmegen (Coq)	GPL
Math Components (Coq+ssreflect)	Not publicly available
Archive of Formal Proofs (Isabelle)	BSD or LGPL
Isabelle Standard Libraries (Isabelle)	BSD
HOL Light Standard Library (HOL Light)	BSD/MIT-like license
Flyspeck (HOL Light)	MIT license?
Wikipedia, PlanetMath	CC-BY-SA
SUMO	GPL + additional restriction
arXiv	CC-BY(-NC-SA) or public domain or only arXiv allowed to distribute

redistribution and changes, but requires maintaining the license and recognizing the contributors.

Table 1 summarises the licenses of several well-known formal mathematical libraries, together with some major informal ones like PlanetMath, Wikipedia, and arXiv.

Also note that the licensing differences between the formal libraries can already now cause nontrivial problems. For example, the Coq Repository at Nijmegen (CoRN) is an advanced mathematical library, which however contains also a number of items that can be generally useful to any Coq formalization. Thus, a credible scenario is that pieces of CoRN might be gradually moved to the Coq Standard Library (distributed with the Coq system). That however is not automatically possible, since CoRN is licensed under GPL, which is stronger (more restrictive) than LGPL (used by the Coq Standard Library). Similar situation might arise when moving the LGPL-licensed entries in the Isabelle Archive of Formal Proofs (AFP) to the Isabelle standard libraries (BSD). So while our initial idea was to possibly optimise the MML license(s) also with respect to possible future transfers and translations between various formal and informal libraries, a survey of the current situation revealed that this is hardly possible in the existing chaos. This again leads us to the necessity of copyright centralization, in order to be able to adapt to the likely future changes of this chaotic global state of affairs.

4 Issues and Their Solutions

In this section we discuss some of the problems we faced when designing a license and copyright mechanism for the MML and how we addressed them.

4.1 Features Required from the License

For the number of reasons mentioned above we wanted our solution for the MML copyright/license to give the SUM some control over the current and future

licensing, while at the same time not hindering legitimate "open science" use of the MML, such as:

- translating its contents so that another proof assistant can use them;
- archiving the MML to record the state of human knowledge;
- allowing MML to be used for data-mining, mathematical search engines, and general AI systems;
- benchmarking ATPs;
- writing formal mathematics and publishing articles about it.

At the same time, we did want the SUM to have the authority to object to uses of the MML that do not adhere to the rules of "open science" and block the free flow of ideas and results contained in the library.

Our solution was to adopt a fairly restrictive open license, adhering to strong copyleft principles, with SUM as a rather powerful central copyright owner. We lay out a policy of "ask and you shall be allowed" when it comes to certain uses of the MML that do not adhere to our restrictive license, but are within the rules of "open science".

4.2 Linking and Adaptation

One consequence of viewing the MML as a collection of executable code concerns the sensitive issue of *linking*. The MML is composed of a large number of items that one can refer to, not unlike one using a subroutine defined in some external library. Thus, if someone has proved the Jordan curve theorem in some form, one can use this theorem to prove some consequence, such as the four-color theorem. Likewise, one can use earlier definitions (such as the definition of the power set operation or topological spaces) in one's own work. Such usage is analogous to linking by virtue of the fact that one's text is not functional (or even meaningful) in the absence of these earlier definitions, and constitutes a derivative work of the used articles, thereby triggering the share-alike mechanism of the GPL.

On the document side, the CC-BY-SA version 3.0 license [2] share-alike mechanism is triggered by the analogous notion of *adaptation*.

A contribution to the MML naturally triggers the share-alike mechanism of both licenses. However, to dispel any doubt about this, the MML licenses come with a binding interpretation note[13] that states that fact explicitly.

4.3 Why Open-Source Copyleft License?

We settled upon a dual-licensing scheme based on the GPL version 3 [5] and CC-BY-SA version 3.0 [2].

The decision to adopt such a scheme was made with some reservation; a dual-licensing scheme is evidently more complicated than a single license. However, the dual-license aspect does suit our situation nicely owing to the dual nature of the MML as code and text. The intention is that the GNU GPL, which aims

[13] http://mizar.org/version/current/doc/COPYING.interpretation

to cover computer code, suits this aspect of the MML, whereas CC-BY-SA, which is designed to cover texts (among other things) seems more appropriate when the MML is considered as a collection of texts. The dual-license scheme is good for *adapting and copying* parts of MML. For example, it allows to copy a piece of an MML proof into Wikipedia or PlanetMath, which are licensed under CC-BY-SA. In such a case, the person copying automatically chooses to use MML under the CC-BY-SA license. In the same spirit, extraction of Prolog (or other) programs from MML mentioned in Section 2.2 would be covered by GPL. The disadvantage of the MML dual-licensing is that *contributing to* MML gets difficult: contributors have to agree both to GPL and CC-BY-SA. This was a difficult decision, however, once we got that far, it allowed us to think even further and come up (after discussions with the Software Freedom Law Center[14]) with copyright assignment (see Section 4.4) as the best of the bad solutions.

Our licenses feature *strong copyleft protection*: anything derived from one's contribution must be similarly freely redistributable and enhanceable. We believe that such transitivity promotes public contribution, because a contributor can engineer his work safe in the knowledge that his efforts, and future enhancements to it, cannot be taken away from him (or from society), and cannot be exploited for private gain without contributing back to the common pot.

4.4 Why Keep the Copyright Ownership with SUM?

In our licensing model, we require that contributor to the MML assign the copyright to their work to the SUM.

The main risk of mandatory copyright assignment is discouraging potential contributors. But since the MML has been functioning with copyright assignment since its beginnings, the risk is mitigated. The community has already adopted this model, and probably will continue to accept it. At worst, mandatory copyright assignment might stunt the future growth of the community.

To assuage this fear, following the models described in [12,8,18], we recommended that:[15]

- SUM be a relatively transparent association, and that it be open to contributors. One way in which SUM is open is through translation of its statutes into the major languages used for science and technology (currently, English). We also insisted that decisions taken by SUM be open to international members (i.e., not requiring physical Polish presence).
- SUM pledges to maintain free (as in freedom) licensing of the assigned work.
- SUM pledges that any profit made by the SUM from the work be used only for the advancement of science.
- The copyright grant to the SUM is automatically rescinded if the SUM breaks the above pledges.

[14] http://www.softwarefreedom.org/

[15] Note that the last three points are now part of the Mizar Copyright Assignment.

4.5 Enforcing FLOSS

Free/libre licenses had, and to a degree still have, a reputation for being difficult, if possible at all, to enforce, or the expose the licensor to abuse from the licensee. This reputation, in our opinion, comes more from the fact that enforcement (mainly by the FSF) used to happen behind closed doors rather than in a public forum, and the final settlement typically would include a "no shaming" clause that kept the polite fiction that the violator voluntarily complied with the GPL, and never imagined doing otherwise, much less did otherwise. Eben Moglen, the general legal counsel for the FSF, and Richard Stallman, the leader of the FSF, used to say (publicly, during conferences) something to the effect of:

> The reason the GNU GPL has not been "tested in court" is that each time the FSF threatens to sue over a GPL violation, the offender chooses to comply with the GPL rather than go to court. This, in essence, means that their legal counsel estimates their losing in court too probable to risk.

More recently, some of the enforcement has become far more public, and sometimes the public shaming is the main force behind the effort. The pioneer of this change is `gpl-violations.org`, created in 2004 by Harald Welte to give GPL enforcement a faster and more dynamic pace than the FSF's usual way of proceeding [22]: the FSF usually let violators continue their infringements for an interim period of time, while the process of bringing them into compliance was ongoing.

More strongly, far from making the contents of the MML more vulnerable to theft, or limiting the freedom of the Mizar community, adopting a FLOSS license such as the one we settled upon gives the community greater strength. There are cases where having a FLOSS license made a crucial difference. One of the earlier examples is the one of g++, the C++ compiler in GCC, the GNU Compiler Collection [17]:

> Consider GNU C++. Why do we have a free C++ compiler? Only because the GNU GPL said it had to be free. GNU C++ was developed by an industry consortium, MCC, starting from the GNU C compiler. MCC normally makes its work as proprietary as can be. But they made the C++ front end free software, because the GNU GPL said that was the only way they could release it. The C++ front end included many new files, but since they were meant to be linked with GCC, the GPL did apply to them. The benefit to our community is evident.
>
> Consider GNU Objective C. NeXT initially wanted to make this front end proprietary; they proposed to release it as .o files, and let users link them with the rest of GCC, thinking this might be a way around the GPL's requirements. But our lawyer said that this would not evade the requirements, that it was not allowed. And so they made the Objective C front end free software.

It is not inconceivable, as formal methods become more widely used, that analogous cases could arise concerning the use of formalised mathematical knowledge.

4.6 What Are Reasonable Conditions for Copyright Ownership?

We settled for a copyright ownership agreement modeled on the FSFE fiduciary license agreement (FLA) [3]. The FLA "allows one entity to safeguard all of the code created for a project by consolidating copyright (or exclusive exploitation rights) to counteract copyright fragmentation." In our case, the entity is SUM. We opted for this kind of agreement to permit possible future changes of the open-source licenses[16] and also selected commercial activities benefiting scientific progress. There already are projects (e.g., the NICTA L4 project[17]) that are sensitive to future commercial uses, while clearly benefiting the development of formal methods. We chose the FSFE's FLA rather than the similar paperwork used by e.g. the FSF mainly because the FSFE's FLA is specifically written for European jurisdictions.

The main substantive change we made to the FLA is allowing the SUM to sell commercial licenses to the MML, when doing so is beneficial to progress in science and technology, subject to the restriction that proceedings must be used to advance the SUM's goals, namely popularising, propagating and promoting the Mizar language. We see this way of proceeding as a tax levied on people that want to benefit from science's production and advancement, without playing by the rules of science of free interchange of ideas and results. In other words, people that want to use the results of science without contributing their further enhancements back to the common pot. This is somewhat similar to the "polluter pays" principle that is becoming widespread in European and international environmental law.

The fact that we set things up legally so that the SUM is allowed to do so does not mean it has to; if a majority of members (contributors) opposes it, it won't happen and the MML will be, by and large, *unavailable* to people not willing to contribute their enhancements to the common pot.

5 Conclusion and Future Work

In order to produce a suitable copyright and licensing model for the MML, we delved into the question of what formal mathematics truly is. We did not settle on a definitive answer yet, and it may well be that (as often with legal concepts) the existing legal concepts and preconceptions need to be updated as scientific progress is being made. The difference between executable code and (formal) mathematics seems to be extremely tenuous, if it exists at all. It is safe to say that formal mathematical texts straddle a boundary between human readability/consumability and machine readability/consumability. An even deeper question: what is formal knowledge, and what is informal knowledge?

Even when we handle this dilemma by dual licensing, there is no clear winner with respect to the goal of making the MML compatible with as many formal and informal mathematical libraries as possible. The situation in this field seems quite

[16] The Wikipedia relicensing trouble being a strong motivation.
[17] http://ertos.nicta.com.au/research/l4/

chaotic, and we hope that this paper will be of some help to the developers of other formal libraries. In particular, our recommendation in this chaotic situation is to centralise the copyright ownership with trusted user associations, so that the situation can be gradually improved by these bodies.

Although the MML has been licensed in a free way, the programs that operate on these texts remain closed-source. Ideally, both the MML and these programs would be free/libre. The license problem here is simpler than the problem of licensing the MML, since we are dealing simply with programs. The process may end up being gradual, starting with the Mizar parser. It is essential that such a problem be tackled; the lack of any kind of open license for the whole of Mizar (its programs and its library), from a political standpoint of scientific research being done for the general good of humanity and available to all, can push away some potential contributors toward other proof assistants and other libraries[18].

For reasons discussed in Section 2.4, we avoided the problem of patents. This is largely because we are not aware now of how patents could play a role in formal mathematics. In the future, though, we may find that the subject needs to be revisited.

References

1. de Bruijn, N.G.: A survey of the project AUTOMATH. In: Hindley, R., Seldin, J. (eds.) To H.B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalisms, pp. 579–606. Academic Press, London (1980)
2. Creative Commons BY-SA, http://creativecommons.org/licenses/by-sa/3.0/ (last accessed 2011/03/14)
3. Free Software Foundation Europe, Fiduciary License Agreement (FLA), http://fsfe.org/projects/ftf/fla.en.html
4. The FLOSSmole project, http://flossmole.org/
5. Free Software Foundation: The GNU General Public License, version 3 (June 29, 2007), http://www.gnu.org/licenses/gpl.html
6. Licenses – definitions of free cultural works. http://freedomdefined.org/Licenses
7. Various licenses and comments about them, http://www.gnu.org/licenses/license-list.html
8. Hillesley, R.: Copyright assignment–once bitten, twice shy. The H Open (August 6, 2010), http://www.h-online.com/open/features/Copyright-assignment-Once-bitten-twice-shy-1049631.html
9. Howison, J., Conklin, M., Crowston, K.: FLOSSmole: A collaborative repository for FLOSS research data and analyses. International Journal of Information Technology and Web Engineering 1, 17–26 (2006), current data http://flossmole.org/content/using-flossmole-data
10. Jackson, T., Dance, G., Ko, H., Kueneman, A., Bost, K., Meek, M.: IBM's Watson trivia challenge, http://www.nytimes.com/interactive/2010/06/16/magazine/watson-trivia-game.html

[18] For example, the third author chose Coq over Mizar for formalisation of surreal (Conway) numbers [13] over such an issue, although Mizar's set theory base could have provided a more natural framework for surreal numbers than Coq's type theory.

11. van Benthem, L.S.: Checking Landau's "Grundlagen" in the AUTOMATH System, Mathematical Centre Tracts, vol. 83. Mathematisch Centrum, Amsterdam, Netherlands (1979)
12. Kuhn, B.M.: Not all copyright assignment is created equal (February 1, 2010), http://ebb.org/bkuhn/blog/2010/02/01/copyright-not-all-equal.html
13. Mamane, L.E.: Surreal numbers in coq. In: Filliâtre, J.-C., Paulin-Mohring, C., Werner, B. (eds.) TYPES 2004. LNCS, vol. 3839, pp. 170–185. Springer, Heidelberg (2006), doi:10.1007/11617990_11
14. The Mizar Mathematical Library. Browsable online, http://mizar.org/
15. Naumowicz, A.: Conjugate sequences, bounded complex sequences and convergent complex sequences. Formalized Mathematics 6(2), 265–268 (1997)
16. Software, B.D.: Open source resource center–license data, http://www.blackducksoftware.com/oss/licenses (last accessed March 2011)
17. Stallman, R.M.: Free Software Free Society: Selected Essays, 1st edn. ch.15, p. 91. GNU Press (2002); 2nd edn, http://shop.fsf.org/product/free-software-free-society-2/; chapter 15: http://www.gnu.org/philosophy/pragmatic.html
18. Stallman, R.M.: When a company asks for your copyright (September 29, 2010), http://www.fsf.org/blogs/rms/assigning-copyright
19. Urban, J.: Translating Mizar for first order theorem provers. In: Asperti, A., Buchberger, B., Davenport, J.H. (eds.) MKM 2003. LNCS, vol. 2594, pp. 203–215. Springer, Heidelberg (2003)
20. Urban, J.: MPTP 0.2: Design, implementation, and initial experiments. Journal of Automated Reasoning 37(1-2), 21–43 (2006)
21. IBM Watson, http://www-03.ibm.com/innovation/us/watson/
22. Welte, H.: GPL enforcement. talk at FOSDEM 2005, (February 2005), abstract at http://archive.fosdem.org/2005/index/speakers/speakers_welte.html
23. Wheeler, D.A.: High assurance (for security or safety) and free-libre open source software (FLOSS)...with lots on formal methods software verification, http://www.dwheeler.com/essays/high-assurance-floss.html (released 2006-06-02) (updated 2009-11-30)
24. Wheeler, D.A.: Make your open source software GPL-compatible. or else, http://www.dwheeler.com/essays/gpl-compatible.html (released 2002-05-06) (revised 2010-09-14)
25. Wiedijk, F. (ed.): The Seventeen Provers of the World. LNCS (LNAI), vol. 3600. Springer, Heidelberg (2006)

Workflows for the Management of Change in Science, Technologies, Engineering and Mathematics

Serge Autexier[1], Catalin David[1,2], Dominik Dietrich[1], Michael Kohlhase[2], and Vyacheslav Zholudev[1,2]

[1] Safe and Secure Cognitive Systems, German Research Centre for Artificial Intelligence (DFKI), Bremen, Germany
[2] Computer Science, Jacobs University Bremen, Germany

Abstract. Mathematical knowledge is a central component in science, engineering, and technology (documentation). Most of it is represented informally, and – in contrast to published research mathematics – subject to continual change. Unfortunately, machine support for change management has either been very coarse grained and thus barely useful, or restricted to formal languages, where automation is possible. In this paper, we report on an effort to extend change management to collections of semi-formal documents which flexibly intermix mathematical formulas and natural language and to integrate it into a semantic publishing system for mathematical knowledge. We validate the long-standing assumption that the semantic annotations in these flexiformal documents that drive the machine-supported interaction with documents can support semantic impact analyses at the same time. But in contrast to the fully formal setting, where adaptations of impacted documents can be automated to some degree, the flexiformal setting requires much more user interaction and thus a much tighter integration into document management workflows.

1 Introduction

As the Web 2.0 age is dawning for mathematics, more and more *mathematical development* is moving online; not just *publications*. An example of this is the PolyMath site, where upon the recent announcement of a proof of $P \neq NP$, the mathematics community has organized itself in a WiKi and found a significant gap in the proof within two weeks; see [4]. The PlanetMath community which has collaborated on 8500 graduate-level encyclopedia articles over 10 years [20] is another, and also the Mizar community, who have formalized more than 60000 definitions, assertions, and proofs and have machine-checked them over the last 40 years. Finally, the Cornell EPrint Archive [21] has amassed over 660 000 scientific articles over 20 years. The hallmark of all these efforts is that they are massive collaborations by many individuals, distributed widely both geographically and temporally. The first three examples have another characteristic that is becoming more and more important: the knowledge items are interdependent

J.H. Davenport et al. (Eds.): Calculemus/MKM 2011, LNAI 6824, pp. 164–179, 2011.

and mutable (subject to change). The sheer size of the knowledge collections together with the fact that many authors do not even know (of) each other induces consistency and coherence problems. In this situation, the need to integrate the mechanisms for "change management" (CM) into the digital libraries seems obvious. Typically, the documents in the libraries are *flexiformal* (flexibly formal) because they contain semantic annotations at different levels of formality. A good example is an informal, but rigorous statement from a mathematical textbook, which intermixes mathematical formulas (formal representations of mathematical objects) with natural language (informal representations of their relations). Change management makes use of the fact that MKM formats explicitly represent the relations between objects to compute related objects and predict the way changes affect them; see [1, 8, 18] for recent progress in this field.

This paper reports on the experiment of integrating CM into the PLANETARY system, a new flexiformal Digital library system, which we will present in the next section. In Section 3, we describe the information present in the sources by way of an extended example and show how these can be used for change management. In Section 4, we present the *DocTIP* system and the CM procedure it implements, so that we can show the integration from an architectural point of view in Section 5. Section 6 revisits the example from Section 3 to show how the information travels through the systems involved. In Section 7, we discuss related work and Section 8 concludes the paper.

2 The Planetary System

The PLANETARY system (see [3, 14, 19] for an introduction) is a Web 3.0 system[1] for semantically annotated document collections in Science, Technology, Engineering and Mathematics (STEM). The system is based on *semantically annotated documents* together with semantic background ontologies (which we call the **content commons**). This information can then be used by user-visible, semantic services like program (fragment) execution, computation, visualization, navigation, information aggregation and information retrieval. Finally a document player application can embed these services to make documents executable. We call this framework the **Active Documents Paradigm** (ADP), since documents can also actively adapt to user preferences and environment rather than only executing services upon user request.

In our approach, *documents published in the* PLANETARY *system become flexible, adaptive interfaces to a content commons* of domain objects, context, and their relations. The system achieves this by providing embedded user assistance through an extended set of user interactions with documents based on an extensible set of client- and server side services that draw on explicit (and thus machine-understandable) representations in the content commons (see Fig. 1).

The PLANETARY system has been used on the course notes of a two-semester introductory course in Computer Science [6] held at Jacobs University by one

[1] We adopt the nomenclature where Web 3.0 stands for extension of the Social Web with Semantic Web/Linked Open Data technologies.

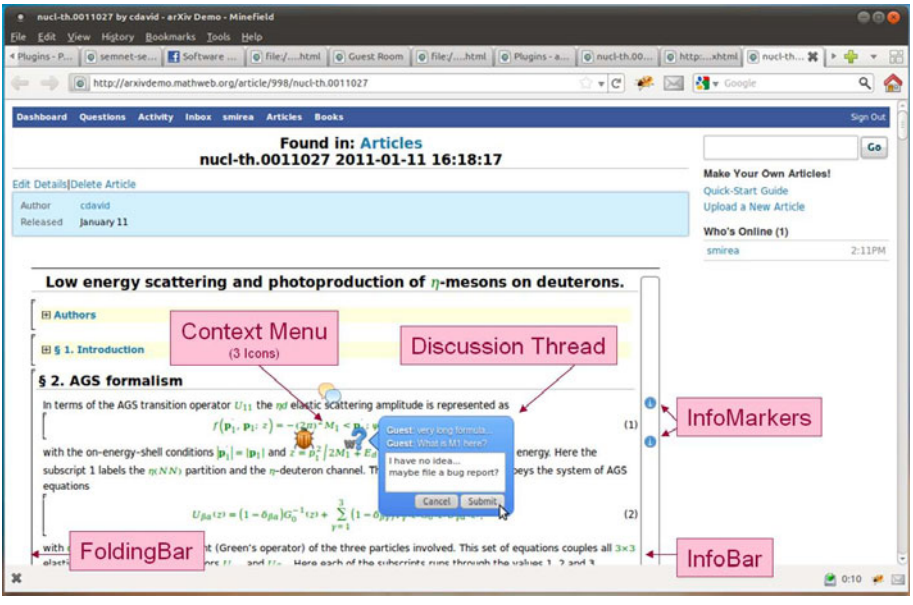


Fig. 1. Course Notes in the PLANETARY system

of the authors in the last eight years. While the basic concept of the course stayed the same over the years, whole topics have been added/moved/deleted, examples and results have been added, and formulations have been sharpened. All of these changes had consequences that were sometimes difficult to foresee, and sometimes led to problematic teaching situations (when the consequences had not been anticipated). The course notes currently comprise 300 pages with over 500 slides organized in over 800 files. This is at the limits of what is manually manageable for the instructor who has authored all of the material; it would be impossible for a new instructor to take over the material (and change it to her liking). It becomes increasingly difficult to manage the over 1000 homework, quiz, and exam problems that have largely been provided by the more than 30 teaching assistants that have accompanied the course over the years.

3 A Planetary Workflow

To get a better intuition for the problems involved in managing changes in flexiformal document collections, consider the situation in Fig. 2 and Fig. 3, which we will use as a running example. The lower part of Fig. 2 shows two well-known definitions from the theory of binary trees and Fig. 3 a lemma that depends on them, as they are referenced in its proof. Clearly, if one of the definitions is changed, then we have to revisit the proof and possibly adapt it or even the lemma to the changed situation.

```
\begin{module}[id=binary-trees]
  \importmodule[\KWARCslides{graphs-trees/en/trees}]{trees}
  \importmodule[\KWARCslides{graphs-trees/en/graph-depth}]{graph-depth}
  ...
  \begin{definition}[id=binary-tree.def,title=Binary Tree]
    A \definiendum[binary-tree]{binary tree} is a \termref[cd=trees,name=tree]{tree}
    where all \termref[cd=graphs-intro,name=node]{nodes}
    have \termref[cd=graphs-intro,name=out-degree]{out-degree} 2 or 0.
  \end{definition}
  ...
  \begin{definition}[id=bbt.def]
    A \termref[name=binary-tree]{binary tree} $G$ is called
    \definiendum[bbt]{balanced binary tree} iff the
    \termref[cd=graph-depth,name=vertex-depth]{depth} of all
    \termref[cd=trees,name=leaf]{leaves} differs by at most by 1, and
    \definiendum[fullbbt]{fully balanced}, iff the
    \termref[cd=graph-depth,name=vertex-depth]{depth} difference is 0.
  \end{definition}
  ...
\end{module}
```

Definition 3.1.7: *(Binary Tree)*
A **binary tree** is a tree where all nodes have out-degree 2 or 0.

Definition 3.1.8: A binary tree G is called **balanced** iff the depth of all leaves differs by at most by 1, and **fully balanced**, iff the depth difference is 0.

Fig. 2. Two definitions and their STEX sources

For humans, it is simple to detect the underlying dependency in principle, but there is a strong possibility that it will be overlooked in practice; especially, if the conceptional distance between a proof and the definitions is large (e.g., because it involves many intervening definitions and assertions). Therefore, authors need system support to keep large mutable knowledge collections in a consistent state. In the situation of our running example, we can make use of the fact that the two text fragments were originally written as semantically annotated STEX course notes [6] for PLANETARY. As such, they contain a lot of semantic annotations that are originally added to drive services like definition lookup, notation adaptation, and just-in-time prerequisites delivery, which also induce a good approximation of the semantic dependency relation that is needed for analysing the impact of changes on definitions and proofs in this and other knowledge items.

Let us consider these annotations in the STEX sources in Fig. 2 and Fig. 3. In the first proof step (the STEX **spfstep** environment) in Fig. 3, the "definition of a binary tree" is referenced, and this reference is marked up by a URI reference encoded in the optional argument of the **premise** macro inside the **justification** element. In the second proof step, the property of being "balanced" is exploited. The fact that the word "balanced" is used as a technical term is marked up with the **termref** macro, whose optional first argument points to the **definiendum** with name bbt in the module binary-trees in Fig. 2.

```
\begin{module}[id=bbt-size]
\importmodule[binary-trees]{binary-trees}
...
```

Lemma 3.1.9 *Let $G = \langle V, E\rangle$ be a balanced binary tree of depth $n > i$, then the set $V_i := \{v \in V : dp(v = i)\}$ of vertexes at depth i has cardinality 2^i.*
Proof: by induction over the depth i

```
  \begin{spfstep}
    By the \begin{justification}[method=byDef]
      \premise[uri=binary-trees,ref=binary-tree.def]{definition of a binary tree}
    \end{justification}, each $\inset{v}{V_{i-1}}$ is a leaf or has
    two children that are at depth $i$.
  \end{spfstep}
  \begin{spfstep}
    As $G$ is \termref[cd=binary-trees,name=bbt]{balanced}
    and $\gdepth{G}=n>i$, $V_{i-1}$ cannot contain leaves.
  \end{spfstep}
  ...
\end{sproof}
\end{module}
```

Fig. 3. A lemma and proof that depend on the definitions in Fig. 2

Intuitively, the relations encoded in these annotations induce the dependency that signals a possible semantic impact of a change to one of the definitions in Fig. 2. There are at least three possible ways an author can benefit from an automated impact analysis based on the semantic annotations in the STEX sources.

C1 An author who wants to change something in one (or both) of the definitions in Fig. 2 can request an estimation of the total impacts costs of a change.

C2 An author who actually changes (one of) the definitions can request an immediate impact analysis, which gives a list of potentially affected knowledge items. This list should be cross-linked to the (presentations of) the affected items, so to simplify navigation. For every item the author will have to decide whether it is really affected and how to adapt it (possibly creating new impacts in the process).

C3 Authors or maintainers of a given knowledge item can be notified of an impact to "their" knowledge item upon changes to elements it depends on.

Note that **C1** and **C2** together constitute what one could call a "push workflow of change management" whereas **C3** corresponds to a "pull workflow". The abundance of semantic references — 12 in this little example — already shows that machine support is indispensable in larger collections. Note furthermore that both of these workflows should be completely independent of the "commit policies" of the knowledge collection. The change management subsystem should support committing partially worked off impact lists — e.g., for the weekend or to pass them on to other authors.

4 *DocTIP*

The *DocTIP* system [5] provides a generic framework that combines sophisticated structuring mechanisms for heterogenous formal and semi-formal documents with an appropriate change management to maintain structured relations between different documents. It is based on abstract document models and abstract document ontologies that need to be instantiated for specific document kinds, such as OMDoc. The heart of the system is the *document broker*, which maintains all documents and provides a generic update and patch-based synchronisation protocol between the maintained documents and the connected *components* working on these documents. Components can be authoring (and display) systems, or analysis and reasoning systems offering automatic background processing support, or simply a connection to a repository allowing to commit and update the documents.

If the document broker obtains a change for some of its documents, the changes are propagated to all connected components for that document. A configurable impact analysis policy allows the system designer to define if impact analysis is required after obtaining a change from some component. To perform the impact analysis the document broker uses the *GMoC*[2] tool ([1] see below) to compute the effect of the change on all documents maintained by the document broker. The *GMoC* tool returns that information as impact annotations to each individual document, which are subsequently distributed to all connected components by the document broker.

4.1 Change Impact Analysis

The key idea to design change impact analysis (CIA) for informal documents is the *explicit semantics method* which represents both the syntax parts (i.e., the documents) and the intentional semantics contained in the documents in a single, typed hyper-graph (see [1] for details). Document type specific graph rewriting rules are used to extract the intentional semantics of documents and the extracted semantic entities are linked to their syntax source, i.e. their *origin*. That way, any change in the document results in semantic objects for which origins have been deleted or changed, as well as syntax objects for which there does not exist corresponding semantic entities yet. The semantic objects are marked with this status information ("deleted", "added", "preserved"). This information is then exploited by analysis rules to compute the ripple effects of the changes on the semantics entities, which in a final stage are used to annotate the syntax parts, that is the documents. The *GMoC* tool is built on top of the graph rewriting tool *GrGen.NET* [10] and is parameterized over document type specific document meta-models and graph rewriting rule systems to extract the semantics and to analyze the impact of changes.

Document Meta-Models. To provide change impact analysis for PLANETARY, we developed a document meta model and graph impact analysis rules for OMDoc.

[2] *GMoC*: Generic Management of Change.

The document meta model consists of a lightweight ontology of the relevant semantic concepts in OMDoc documents, — e.g., theories, symbol declarations and their occurrences, axioms, definitions, assertions, and their use in proofs and proof steps — together with semantic relations between concepts — e.g., import relations between theories, symbols and their definitions, assertions and their proofs. Note that the OMDoc meta-model abstracts over the OMDoc surface syntax. For instance, a definition can either be a definition-element

```
<symbol name="unit">
<definition xml:id="mon-d1" for="unit" type="informal">
 <CMP>
  A structure (M,*,e), in which (M,*) is a semi-group with unit e is called monoid.
 </CMP>
</definition>
```

where the symbol defined by the definition is given by the for attribute of the definition (boxes abbreviate OpenMath content here). The symbol itself is declared in a different element. This kind of definition typically occurs when OMDoc documents are created manually or obtained from formal representations. Alternatively, a definition can come as a "typed" omtext such as

```
<omtext type="definition" xml:id="binary-tree.def" about="#binary-tree.def">
 <CMP xml:id="binary-tree.def.CMP1" about="#binary-tree.def.CMP1">
  <p xml:id="binary-tree.def.CMP1.p1" about="#binary-tree.def.CMP1.p1">
   A <term cd="balanced-binary-trees" name="binary-tree" role="definiendum">
   binary tree</term> is a <term cd="trees" name="tree"
   xml:id="binary-tree.def.CMP2.p1.term2"
   about="#binary-tree.def.CMP2.p1.term2">tree</term> where all ...
  </p>
 </CMP>
</omtext>
```

which typically happens, for instance, when generating the OMDoc files from an LaTeX source file. Note that in this case the defined symbol is declared by the term element with role="definiendum". The fact that this definition defines that symbol comes from the structural nesting of the term inside the definition. Similar examples are theories which can either be imported into each other by using the explicit imports elements or simply by nesting theory-elements.

Conceptually, it does and should not matter in which form symbols and definitions are given, and a mixture of both forms is also desirable to support the linking of mathematical content in OMDoc from different authoring sources. The document meta model declares these pure concepts and relations like an ontology. The intentional semantics of a given OMDoc document is a set of instances of these concepts and relations. The used graph rewriting tool supports hypergraphs with typed nodes and edges. The types are simple types with subtyping relations. This is exploited to subdivide the whole graph in a syntax and a semantic subgraph by introducing top-level types for either part. The OMDoc syntax elements are declared as subtypes of the syntax type and the OMDoc document being an XML tree can then naturally be represented as (syntax)

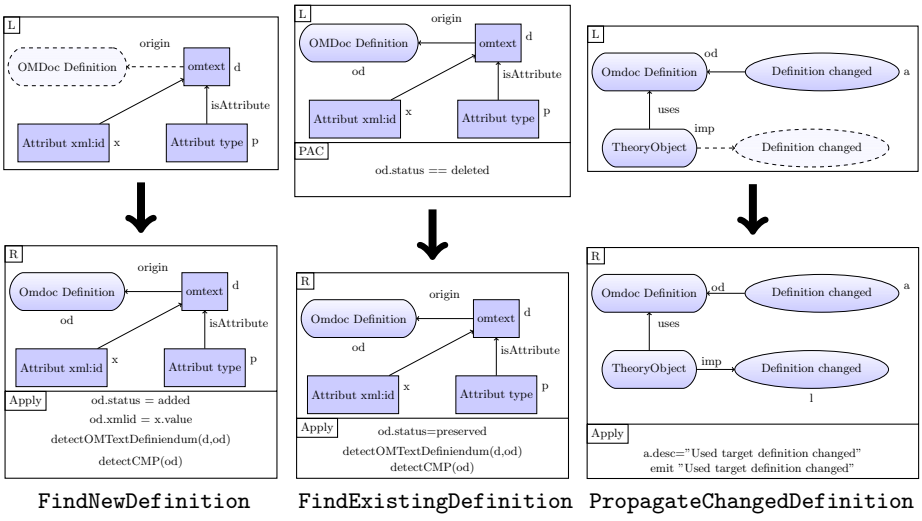


Fig. 4. Two Abstraction Rules and one Propagation Rule written top-down; we use rectangles for syntax nodes, rounded rectangles for semantic nodes and ellipses for impact nodes.

nodes and relations. Analogously, the semantic concepts and relations from the OMDoc document meta-model are simply declared as subtypes of the semantic types.

Abstraction Phase of CIA. The abstraction phase of the impact analysis for OMDoc documents consists of extracting the intentional semantics from the given OMDoc documents. This is realized by a set of graph rewriting rules which analyse the OMDoc document to extract the semantic concepts and relations, and mark them as being added. Examples of such rules are the two left-most rules in Fig. 4 to extract definitions from "typed" `omtext`: The graph rewriting rules are named (e.g., `FindNewDefinition`) and have a left-hand side (the box labelled by `L`) indicating the pattern to match in a subgraph and a right-hand side (the box labelled `R`) by what the instantiated subgraph pattern is replaced. Identical graph nodes and edges are additionally labelled by names, such as $x, d, p, od, \ldots$. Further conditions that must be satisfied to enable the graph rewriting step are positive application conditions (PAC), which must hold on the graph before rule application and negative application conditions (the dashed nodes and edges in the left-hand sides `L` or in extra NAC boxes—not used here) which must be false on the graph before rule application. These conditions can be graph patterns as well as boolean tests on attribute values. The application of the graph rewriting rule replaces the subgraph in `L` with the subgraph in `R` and additional adaptations can be triggered in the `Apply` part, such as adapting the value of attributes but also invoking further graph rewriting rules using their name (e.g., `detectCMP`).

The rules for the abstraction phase always come in two variants: one variant is for new syntactic `omtext`s, i.e., there does not exist yet a semantic object in

```
<impacts>
  <impact for="binary-tree.def" name="Definition changed"
          select="⟨xpath-to-definition-binary-tree⟩"/>
  <impact for="balanced-binary-tree.def" name="Definition Binary Tree changed"
          select="⟨path-to-definition-balanced-binary-tree⟩"/>
  <impact for="size-lemma-pf.derive2.method2.proof2.derive4.CMP1.p1.term2"
          name="Definition Binary Tree changed"
          select="⟨path-to-inproof-reference-to-balanced-binary-tree⟩"/>
</impacts>
```

Fig. 5. Example Impact Annotation File

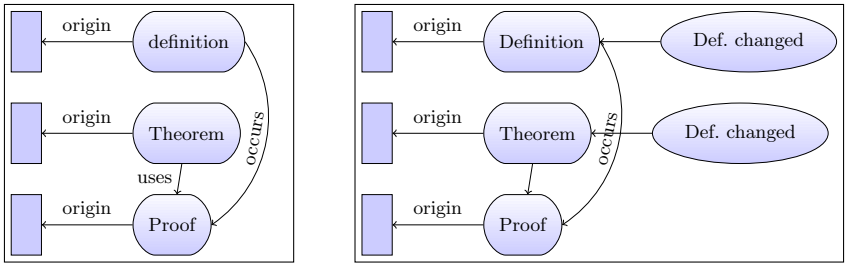
the semantic graph. For these, new semantic instances are introduced, marked as added and the origin of the semantic concept is represented explicitly by an Origin edge from the semantic node to the syntax node. The second variant is for already known syntactic omtexts, i.e., there exist already a semantic object in the semantic graph from a previous version of the document. For these, the semantic instances are maintained and marked as preserved. Both rules invoke further rules to analyse the "body" of a definition in order to find out whether the definition has changed (e.g., detectCMP). All semantic objects that are neither added nor preserved are marked as deleted by a generic rule operating over all semantic nodes and edges. Overall we have designed 91 rules for the abstraction phase that synchronizes OMDoc documents with their intentional semantics.

Propagation Phase of CIA. The second, so-called *propagation* phase, analyses the semantic graph and exploits the information about semantics objects and relations being marked as added, deleted or preserved to propagate the impact of changes through the semantic graph. Impacts are a third type of nodes, different from the syntax and semantic nodes. They contain a human-oriented description of the impact and can only be connected to semantic nodes. For instance, we have one marking a definition for some symbol, say f, as being changed, when its body has changed. Furthermore, we have rules that propagate that information further to definitions that build upon f or proofs using that definition (see rightmost rule of Fig. 4 for an example). Overall, we have 15 rules to analyse and propagate the impacts.

Projection Phase of CIA. Finally, we have the *projection* phase which essentially consists of one generic rule that projects the impact information of the semantic nodes backwards along the origin links to the syntactic node and creates a corresponding impact annotation for the syntactic part of the documents. The impact annotations are output in a specific XML format, where an impact annotation refers to the xml:id of the OMDoc content element in its for attribute and the name attribute contains the human-oriented description of the impact. For our running example we obtain the impact shown in Fig. 5.

4.2 Change Impact Analysis Workflow

The workflow inside *DocTIP* for the change impact analysis is to initially build up a semantics graph for all documents that shall be watched by *DocTIP*. For



(a) Initial Syntax and Semantics (b) Propagated Impacts after Definition Change

Fig. 6. Change Impact Analysis Phases

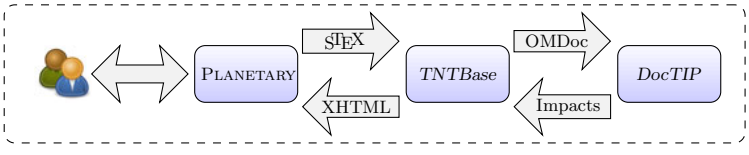


Fig. 7. Tool Chain

our running example, the relevant parts of the initial graph are given in Fig. 6(a). Upon a change in some document, a semantic difference analysis between the old and the new OMDoc documents is performed, which results in a minimal change description on an appropriate level of granularity. This is also provided by the *GMoC* tool, which includes a generic semantic tree difference analysis algorithm parameterized over document-type specific similarity models. The computed changes are applied to the syntactic document graph. Subsequently, the impact analysis rule systems of the three phases *abstraction*, *propagation*, and *projection* are applied exhaustively in that order[3]. For our running example, we obtain a graph of the form in Fig. 6(b). As a result of the impact analysis, the *DocTIP* system returns the computed impacts in the XML format described before for *all* documents it is maintaining (not only for the document that caused the change).

5 System Architecture

In order to add change management support for the workflows, we consider the architecture of the PLANETARY system (see Fig. 7). The user interacts with the PLANETARY system via a web browser, which presents the mathematical knowledge items based on their XHTML+MathML presentation in a WiKi-like form.

[3] Termination must be ensured by the designer of the rules systems. However, the *GrGen.NET*-framework comes with a strategy language, that allows for a fine-grained control over the rule executions, which helps a lot for designing the strategies of the different phases. It is also used to sequentialize the three phases.

The XHTML+MathML documents are rendered from content oriented mathematical knowledge items in OMDoc format. Along with the XHTML+MathML document versions, the PLANETARY system maintains the original STEX document snippets, which the author can edit in the web browser. The OMDoc documents are maintained in the *TNTBase* repository together with their original STEX source snippets. Any change in the OMDoc documents in *TNTBase* results in an update of the corresponding knowledge items in the PLANETARY system after rendering the OMDoc in XHTML+MathML. Upon edit of the STEX snippets in the PLANETARY system, a new OMDoc is created from the STEX sources [7] and pushed into *TNTBase*, which returns the XHTML+MathML presentation.

The *TNTBase* [24] is a Subversion based repository for normal files as well as XML files. It behaves likes a normal Subversion repository, but offers special support for XML documents by storing the revisions in a XML database. By this it allows efficient access via XQueries to XML objects. *TNTBase* allows the definition of document specific presentation routines, such as the XHTML+MathML rendering of OMDoc documents. For its role as repository for the PLANETARY system, it is important to note that the STEX snippets and the corresponding OMDoc documents are stored together in the same directory in *TNTBase*, such as, for instance,

- the file balanced−binary−trees.tex that contains the source of Fig. 2, and
- balanced−binary−trees.omdoc that contains the OMDoc transformation.

To add change management support, we connected *DocTIP* to *TNTBase*. *DocTIP* returns impact information in form of annotations to the OMDoc documents, which are stored in the *TNTBase* as an extra file together with the OMDoc and the STEX files, but with the extension ".imp", such as

- balanced−binary−trees.imp (in the XML format shown in Fig. 5).

Like the change in the OMDoc file, any change in the impacts file is forwarded as is by *TNTBase* to the PLANETARY system. The rendering of OMDoc in XHTML+MathML preserves the xml:id. Therefore, the PLANETARY assigns the impacts to the XHTML+MathML snippets using the for-attributes and presents on the WiKi-page.

6 Example Revisited

To see how the parts of the system interact, let us revisit the example from Section 3. Say the user found a typo in the binary trees module in Fig. 2. She opens the web editor, corrects it and submits the changed module (see Fig. 8; note that the user requested a change impact analysis). The system communicates the changes to *DocTIP*, which determines the list of impacts based on the semantic relations. *DocTIP* in turn communicates the impacts to *TNTBase*, which stores them for further reference and passes them on to the PLANETARY system. Moreover, it notifies the user about impacts by updating

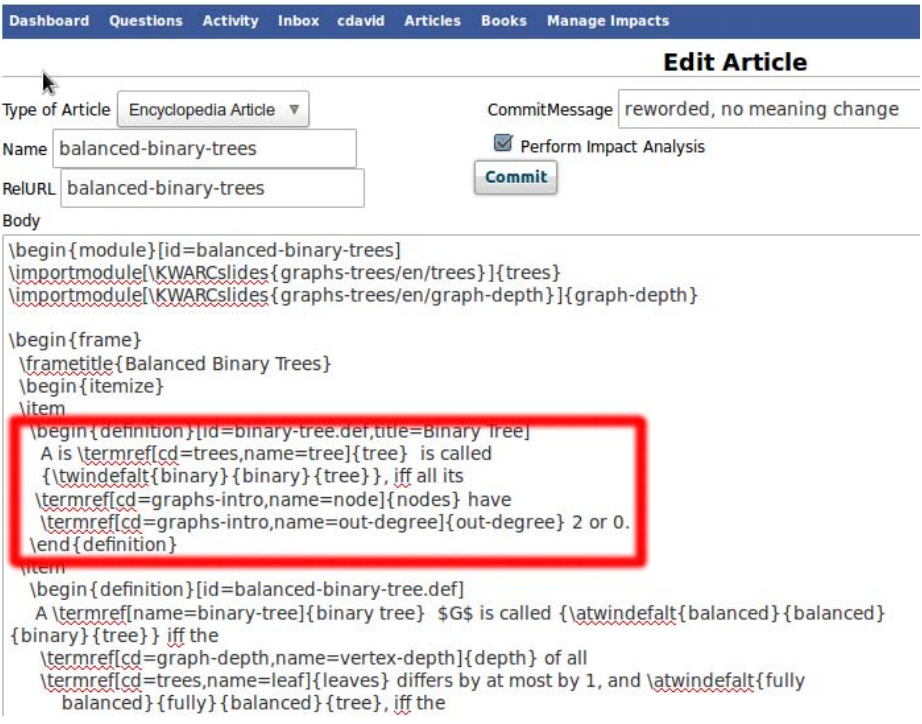


Fig. 8. Committing Changes in PLANETARY

the superscript number on the "Manage impacts" field in the top menu bar (see Fig. 9). If the user decides to act on the impacts, she gets the impact resolution dialog in Fig. 9, which has a tab for every module that is impacted by the change. Note that the user gets the module in its presented form as this is the most readable view, and, furthermore, we can use the identifiers in the impacts (see Fig. 5) to highlight the affected objects. For each of them, the user can then either discard the impact information if it was a spurious impact (via the checkmark icon in the "Accept Change" box) or edit the source of the impacted object (via the "edit" icon in the box) and mark it as resolved afterwards. Alternatively, she can use the action links above to make changes at the level of the whole module. Note that a conventional conflict resolution dialog via three-way merge as we know it from revision control systems does not apply to this situation, since we only have to deal with "long-range conflicts", i.e., impacts between different objects, not conflicting changes to a single object. When the user quits the impact resolution dialog, all discarded and resolved impacts are communicated to *TNTBase* together with the changes. *TNTBase* updates the set of tabled impacts and communicates the changes to *DocTIP* for a further round of CIA. Note that the storage of tabled impacts in *TNTBase* (the additional ".imp" files) makes the change management workflow more flexible over time. The need for

Fig. 9. The Impact Resolution Dialog

this was unanticipated before the integration and triggered a re-design of the system functionality.

7 Related Work

There exists several methods for software development that estimate the scope and complexity of a change of a piece of software with respect to other modules and documentation, known as *software change impact analysis*. The methods are usually based on modeling data, control, and component dependency relationships within the set of source code. Such relationships can be automatically extracted using well-known techniques such as *data-flow analysis* [23], *data dependency analysis* [12], *control flow analysis* [16] , *program slicing* [15], *cross referencing* and *browsing* [2], and logic-based *defects detection* and *reverse engineering algorithms* [9]. From an abstract point of view, we have a similar set-up as we extract and collect relevant information and their dependencies in the semantical extension of the document. For example, the process of extracting dependencies between definitions, axioms, and theorems and their uses in proofs can be seen to be similar to a data-flow or dependency analysis for software. However, on the concrete level, our approach differs because the flexiformal documents we deal with do not have a formal semantics as software artefacts. Indeed, we cannot directly interpret the textual parts of STEX documents, but have to rely on the STEX markup manually provided by the author. Thus, the impact

analysis can always only be as accurate as the manual annotations are. Furthermore, not having a formal semantics at hand, we cannot automatically check if a certain change really has an impact on other parts. In order to be "complete", we have to follow a possibilistic approach to propagate impacts and thus may get false positives, i.e., spurious impacts. Since impact information for some parts may trigger further impact propagations (due to the possibilistic approach), this may result in many spurious impacts in principle. For this a dependency management on impacts nodes themselves can be used (by adding dependency links between impacts in the rules) in order to propagate the deletion of spurious impact information by the user.

Requirement tracing [11] is the process of recording individual requirements, linking them to system elements, such as source code, and tracing them over different levels of refinement. Several tools have been developed to support requirement tracing, such as the *Doors* system [17]. Within our setting, the change of an object, e.g., a definition, gives rise to an impact, such as to revise the proof of a theorem. Similar to requirements, these impacts are linked to concrete objects and may depend on each other. Also similar is that requirements are formulated in natural language and the requirements tracing system has no access to the semantics, hence also has to follow a possibilistic approach. Of course, the type of relationships between requirements are tailored to that domain in requirement tracing as they are in our case. The main difference is that with our approach the relationships are not built into the tool, but can be defined externally in separate rule files. This allows the addition of new relationships, types of impacts and propagation rules, for instance in order to accommodate the various extensions of OMDoc like for exercises, but also for didactic knowledge. This will enable to add change impact analysis for E-learning systems like ActiveMath [22], that are based on OMDoc with their own didactic extensions and that lack change impact analysis support for the authors of course materials.

8 Conclusion

We have presented an integration of a management of change functionality into an active document management system. The combined system uses the semantic relations that were originally added to make documents interactive to propagate impacts of changes and ultimately help authors keep the collections of source modules consistent. The approach is based on impact analysis via graph rewriting rule systems for a core of the OMDoc format. CIA support for extensions of that core OMDoc can easily be added on demand and, due to the generic nature of impact descriptions and their handling in PLANETARY, the presentation module does not need to be adapted.

One limitation of the current integration that we want to alleviate in the near future is that our integration currently assumes a single-user mode of operation, as we have no means yet to consistently merge the three kinds of documents (STEX, OMDoc and impacts file). Moreover, multiple users working on different branches that are partly merged on demand are also not supported yet. One of

the main conceptual problems to be solved here is how to deal with propagating changes by "other authors". For that we plan to build in the notion of versioned links proposed in [13]. Finally, the current policy to eagerly trigger the change impact analysis after each edit may be undesirable in situations where the author wants to perform several small edits, which currently may result in many spurious impacts. The impact analysis policy is pre-configured in *DocTIP* and we could easily enable the author to change it via the PLANETARY system. However, it requires a mechanism to enforce impact analysis eventually, in order to prevent to just turn it off. For this we need to gather more experience what would be a suitable policy to optimally fit into the workflows.

Acknowledgements. We would like to thank the anonymous reviewers for their feedback and Andrea Kohlhase who semantically annotated manually an extended document corpus for discussions about the presented workflows and her comments on earlier versions of this paper.

References

1. Autexier, S., Müller, N.: Semantics-Based Change Impact Analysis for Heterogeneous Collections of Documents. In: Gormish, M., Ingold, R. (eds.) Proceedings of 10th ACM Symposium on Document Engineering (DocEng 2010), Manchester, UK (2010)
2. Bohner, S.A.: A graph traceability approach for software change impact analysis. PhD thesis. George Mason University, Fairfax, VA, USA (1995)
3. David, C., et al.: eMath 3.0: Building Blocks for a social and semantic Web for online mathematics & ELearning. In: Mierlus-Mazilu, I. (ed.) 1st International Workshop on Mathematics and ICT: Education, Research and Applications, Bucharest, Romania (November 3, 2010)
4. Deolalikar P vs NP paper, http://michaelnielsen.org/polymath1/index.php?title=Deolalikar_P_vs_NP_paper&oldid=3654 (visited on 11/03/2010)
5. DocTIP: Document and Tool Integration Platform, http://www.informatik.uni-bremen.de/agbkb/forschung/formal_methods/DocTIP/ (visited on 11/30/2010)
6. General Computer Science: GenCS I/II Lecture Notes. Semantic Course Notes in Panta Rhei (2011), http://gencs.kwarc.info/book/1
7. Ginev, D., Stamerjohanns, H., Kohlhase, M.: The LATEXML Daemon: Editable Math on the Collaborative Web. Intelli- gent Computer Mathematics (accepted, 2011)
8. Hutter, D.: Semantic Management of Heterogeneous Documents (Invited Talk). In: Aguirre, A.H., Borja, R.M., Garciá, C.A.R. (eds.) MICAI 2009. LNCS, vol. 5845, pp. 1–14. Springer, Heidelberg (2009)
9. Hwang, Y.-F.: Detecting faults in chained-inference rules in information distribution systems. PhD thesis. George Mason University, Fairfax, VA, USA (1998)
10. Jakumeit, E., Buchwald, S., Kroll, M.: GrGen.NET. International Journal on Software Tools for Technology Transfer (STTT) 12(3), 263–271 (2010)
11. Jarke, M.: Requirements Tracing. Communication of the ACM 41(12) (1998)
12. Keables, J., Roberson, K., von Mayrhauser, A.: Data Flow Analysis and its Application to Software Maintenance. In: Proceedings of the Conference on Software Maintenance, pp. 335–347. IEEE CS Press, Los Alamitos (1988)

13. Kohlhase, A., Kohlhase, M.: Maintaining Islands of Consistency via Versioned Links
14. Kohlhase, M., et al.: The Planetary System: Web 3.0 & Active Documents for STEM. Accepted for publication at ICCS 2011 (Finalist at the Executable Papers Challenge) (2011)
15. Korel, B., Laski, J.: Dynamic slicing of computer programs. The Journal of Systems and Software 13(3), 164–1212 (1990), http://dx.doi.org/, doi:10.1016/0164-1212, ISSN: 0164-1212
16. Loyall, J.P., Mathisen, S.A.: Using Dependence Analysis to Support the Software Maintenance Process. In: ICSM 1993: Proceedings of the Conference on Software Maintenance, pp. 282–291. IEEE Computer Society, Washington, DC, USA (1993) ISBN: 0-8186-4600-4
17. rcm2 Ltd. DOORS - Dynamic Object-Oriented Requirements System, http://www.rcm2.co.uk
18. Müller, N.: Change Management on Semi-Structured Documents. PhD thesis. Jacobs University Bremen (2010)
19. Planetary Developer Forum, http://trac.mathweb.org/planetary/ (visited on 01/20/2011)
20. PlanetMath.org Math for the people, by the people, http://planetmath.org (visited on 01/06/2011)
21. arxiv.org e-Print archive, http://www.arxiv.org (visited on 01/08/2010)
22. The ActiveMath System, http://www.activemath.org/ (visited on 03/11/2011)
23. White, L.J.: A Firewall Concept for both Control- Flow and Data- Flow in Regression Integration Testing. IEEE Trans. on Software Engineering, 171–262 (1992)
24. Zholudev, V., Kohlhase, M.: TNTBase: a Versioned Storage for XML. In: Proceedings of Balisage: The Markup Conference 2009. Balisage Series on Markup Technologies. Mulberry Technologies, Inc., (2009), http://kwarc.info/ vzholudev/pubs/balisage.pdf

Parsing and Disambiguation of Symbolic Mathematics in the Naproche System

Marcos Cramer, Peter Koepke, and Bernhard Schröder

University of Bonn and University of Duisburg-Essen
{cramer,koepke}@math.uni-bonn.de, bernhard.schroeder@uni-due.de
http://www.naproche.net

Abstract. The Naproche system is a system for linguistically analysing and proof-checking mathematical texts written in a controlled natural language. The aim is to have an input language that is as close as possible to the language that mathematicians actually use when writing textbooks or papers.

Mathematical texts consist of a combination of natural language and symbolic mathematics, with symbolic mathematics obeying its own syntactic rules. We discuss the difficulties that a program for parsing and disambiguating symbolic mathematics must face and present how these difficulties have been tackled in the Naproche system. One of these difficulties is the fact that information provided in the preceding context – including information provided in natural language – can influence the way a symbolic expression has to be disambiguated.

Keywords: Symbolic mathematics, mathematical formulae, Naproche, formula parsing.

1 Introduction

In recent years, formal mathematics has seen remarkable progress. Proofs of significant mathematical theorems like the Jordan Curve Theorem or the Prime Number Theorem have been formalized and formally checked in powerful systems like HOL light [10] [1]. Some large scale formalization projects related to current research mathematics are under way [11].

This creates a demand for presenting developments in formal mathematics within the ordinary language used by mathematics. De Bruijn [3], one of the pioneers of formal mathematics, formulated :

> [...] the Automath project tries to bring communication with machines in harmony with the usual communication between people.

He describes his approach as follows:

> So I got to studying the structure of mathematics by starting from the existing mathematical language and from the need to make such language understandable for machines. I think we might call that approach "natural". "Natural deduction" is a part of it. [...] the word "natural" [...] refers to the reasoning habits of many centuries, [...]

J.H. Davenport et al. (Eds.): Calculemus/MKM 2011, LNAI 6824, pp. 180–195, 2011.

The Naproche project[1] (NAtural language PROof CHEcking) tries to take up some of the challenges of natural formal mathematics. Due to the tremendous difficulties in in the deep semantic analysis of natural language and common (mathematical) arguments, known from linguistics and artificial intelligence, at the moment, success can only be limited: the general problem appears to be "AI-hard". Naproche therefore restricts itself to a kind of existence proof: to show that one can formulate substantial mathematical texts, so that they are acceptable texts for ordinary mathematicians, but simultaneously computer readable and checkable for linguistic and mathematical correctness. This involves several subtasks, in particular the development of a controlled natural language for mathematics with corresponding parsing mechanisms, parsing of (LaTeX-style) mathematical formulae, translations into first-order formats, and the connection with strong automatic provers to supply missing "trivial" proof elements. Currently we are in the process of reformulating parts of E. Landau's *Grundlagen der Analysis* [16] in the Naproche controlled language and simultaneously developing the Naproche formal mathematics system.

In this paper we address the problem of parsing mathematical formulae embedded in some mathematical text. Despite the widespread assumption that mathematical formulae are exact, they are often very ambiguous in a way that (standard) typing does not sufficiently resolve. We study situations in which further information, mathematical and linguistic, from the ambient text has to be taken into account.

We demonstrate with a number of representative examples, that fairly complex formulae, written in "simple LaTeX", can be correctly parsed, and we expect to be able to parse nearly all formulae that will be coming up in the formalization of Landau's *Grundlagen*. Unfortunately, a further evaluation of our methods appears to be problematic at this moment. Due to the many styles of writing mathematics and coding it in LaTeX we cannot hope to be able to parse arbitrary formulae from large repositories of mathematical material. Like with the controlled input language we are dependent on adequate reformulations, where adequacy has to be judged by experts in the subject. Note that reformulations and reformalizations are ubiquitous in formal mathematics anyway, to get proofs to work. To determine "degrees of naturality" is notoriously problematic, as is well-known from experimental linguistics, and has to be left to the reader's appreciation.

After presenting the Naproche System in section 2, we exhibit the flexibility of symbolic mathematics in section 3, explaining why this flexibility makes symbolic mathematics so difficult to parse and disambiguate. In section 4, we proceed to discuss possible approaches to tackling these difficulties. Our solution to these problems is presented in sections 5 to 8, which are followed by a section on related work and a conclusion.

[1] Naproche is a joint initiative of Peter Koepke (Mathematics, University of Bonn) and Bernhard Schröder (Linguistics, University of Duisburg-Essen). The Naproche system is technically supported by Gregor Büchel from the University of Applied Sciences in Cologne.

2 The Naproche System

A central goal of Naproche is to develop and implement a controlled natural language (CNL) for mathematical texts which can be transformed automatically into equivalent formulae of first-order logic using methods of computational linguistics [7]. We have developed a prototypical *Naproche system*, which can automatically check texts written in the Naproche CNL for logical correctness [6]. We test this system by reformulating parts of mathematical textbooks and the basics of mathematical theories in the Naproche CNL and having the resulting texts automatically checked.

The Naproche system transforms a given input text into a logical representation of its content, called a *Proof Representation Structure* (PRS). The PRS is checked for logical correctness with the aid of *automated theorem provers*. More precisely, the PRS creation and checking process is performed sentence by sentence: For every sentence in the text, the system first parses the sentence and updates the PRS accordingly; afterwards, it checks the logical correctness of the additions that have been made to the PRS. The checking algorithm keeps a list of first order formulae it considers to be true, called premises, which gets continuously updated during the checking process, and which represents the mathematical knowledge that a reader of the text has collected up to a given point in the text.

3 Symbolic Mathematics

One of the conspicuous features of the language of mathematics is the way it integrates mathematical symbols into natural language material. The mathematical symbols are combined to *mathematical expressions*, which are often referred to as *mathematical formulae* or *mathematical terms* depending on whether they express propositions or whether they refer to a mathematical object. Already at first sight, a whole variety of syntactic rules are encountered for forming complex terms and formulae out of simpler ones; a basic classification of these was provided by Ranta [18]:

- There are infix operators that are used to combine two terms to one complex term, e.g. the $+$ symbol in $m + n$ or $\frac{1}{x} + \frac{x}{1+x}$.
- There are suffix operators that are added after a term to form another term, e.g. the ! symbol in $n!$.
- There are prefix operators that are added in front of a term to form another term, e.g. sin in $\sin x$.
- There are infix relation symbols used to construct a formula out of two terms, e.g. the $<$ symbol in $x < 2$.

As noted by Ganesalingam [9], "this simple classification is adequate for the fragment Ranta is considering, but does not come close to capturing the breadth of symbolic material in mathematics as a whole." It does not include notations

like $[K : k]$ for the degree of a field extension, it does not allow infix operators to have an internal structure, like the $*_G$ in $a *_G b$ for denoting multiplication in a group G, nor does it account for the common way of expressing multiplication by concatenation, as in "$a(b + c)$".

Another kind of prefix operator not mentioned by Ranta is the one that requires its argument(s) to be bracketed, e.g. f in $f(x)$. (Of course, the argument of a prefix operator like sin might also be bracketed, but generally this is done only if the argument is complex and the brackets are needed for making sure the term is disambiguated correctly.) This is even the standard syntax for applying functions to their arguments, in the sense that a newly defined function would be used in this way unless its definition already specifies that it should be used in another way.

The expression $a(x + y)$ can be understood in two completely different ways, depending on what kind of meaning is given to a: If a is a function symbol and $x+y$ denotes a legitimate argument for it, then $a(x+y)$ would be understood to be the result of applying the function a to $x+y$. If on the other hand a, x and y are – for example – all real numbers, then $a(x + y)$ would be understood as the product of a and $x+y$. Now whether a is a function or a real number should have been specified (whether explicitly or implicitly) in the preceding text. This is why we say that the disambiguation of symbolic expressions requires information from the preceding text, and this information might have been provided in natural language rather than in a symbolic way.

As one can already see from these sketches of symbolic mathematics, the task of parsing and disambiguating symbolic expressions has a lot of aspects.

One of the issues that has to be surmounted in order to treat mathematical symbolism directly in a computer program is its two-dimensionality. Mathematicians extensively use superscripts and subscripts and put terms above other terms as in the fraction notation. Naproche has already for some time adopted LaTeX for its input, so that in this paper we restrict ourselves to parsing and disambiguating the LaTeX code that is used for generating mathematical formulae[2]. The reversion of a pictorial symbolic input into a LaTeX input or another linearisation of it is certainly an interesting undertaking, but outside the scope of this paper.

In order to cope efficiently with the diversity of possible LaTeX codes for a given symbolic output – e.g. `a^b` and `a^{b}` both producing a^b – we normalise the LaTeX input before the actual parsing process, in this case to `a^{b}`. For the rest of this paper, we use this normalised LaTeX code whenever it is necessary for the explanation; when the LaTeX code is not necessary for the explanation, we use the typographic notation that depicts the mathematical symbols as they are commonly drawn and printed.

[2] We restrict ourselves to standard LaTeX, i.e. without any user-defined macros. Additionally, we in some respects require the author to use *neat* LaTeX, e.g. to write the sine function using `\sin` rather than `sin` in order to distinguish it from the concatenation of the three variables s, i and n.

4 Possible Approaches to Disambiguation

If $a(x + y)$ is to be read as the value of a function a at $x + y$, then a has to be a function. This requirement can be understood in two different ways, which are nevertheless related and combinable: Either it is considered to be a *presupposition* of the symbolic expression $a(x + y)$; in this case, the linguistic theory of presuppositions with all its elaborations might be considered to be applicable to this case [13] [8]. Or it is considered to be a *type judgement* about a; in this case, it should be possible to formulate a type system for symbolic mathematics and reuse existing ideas from type theory to describe and work with this type system.

In the context of a proof checking system like Naproche, presuppositions have to be checked for their correctness, i.e. the presuppositions of an expression have to be checked to logically follow from the premises that are available at the point where the expression is used [8]. One possible approach that we took into consideration for disambiguating symbolic expressions was to check their presuppositions already during the parsing process, so that readings which lead to wrong presuppositions would already be blocked during the parsing process. This approach, however, has turned out to be far too inefficient: It would involve constantly calling automatic theorem provers during the parsing process and waiting for their output before continuing the parsing.

Another approach is to rely on a type system rather than on presupposition fulfillment for disambiguating symbolic mathematics. In that case, one needs a very rich and flexible type system for symbolic mathematics. Such a type system has been developed ingeniously by Ganesalingam [9]. However, to attain the richness of the type system required for handling all kinds of ambiguities that can arise, he was obliged to require the author of a text that is to be parsed by his system to write sentences whose sole function is to create types that are needed for certain disambiguations. Given that the goal of Naproche is to stay as close as possible to the language that mathematicians naturally use, this aspect of Ganesalingam's approach made it less attractive for us.

So we decided to take up a combined approach, in which there is a relatively simple type system capable of blocking most unwanted readings during the parsing process, with the remaining readings being filtered by checking their presuppositions.

5 A Type System for Symbolic Mathematics

In the type system that Naproche uses for handling symbolic mathematics, there are two basic types: i for individuals and o for formulae expressing propositions. Apart from these, there are function types of the form $[t_1, ...t_n] \to t$, where $t_1, \ldots, t_n$ are the types of the arguments the function takes and t is the type of the term that we get when we apply this function to legitimate arguments. So unlike in the Simple Theory of Types (STT) [5], we have an inherent way of handling multi-argument functions. In STT, multi-argument functions must

be simulated by functions whose codomain type is again a function type, e.g. + would be considered a function from natural numbers to functions from natural numbers to natural numbers. We, however, want to use types to describe how mathematical formulae are structured in actual mathematical texts, and for this purpose it is better to have multi-argument functions inherently in the type system.

Note that formulae are also considered terms (namely terms of type o), and that the logical connectors are considered functions of type $[o, o] \to o$ or $[o] \to o$. Even quantifiers are considered to be functions, namely two-place functions whose first argument has to be a variable and whose second argument is a term of type o that may depend on the variable. We formalise this by writing the type of quantifiers as $[var(\_, X), X - o] \to o]$, where $var(\_, X)$ means that the first argument is a variable X of type $\_$ (i.e. of any type), and $X - o$ means that the second argument is a term of type o possibly depending on X[3].

5.1 Syntactic Types

As already discussed in section 3, functions can behave in syntactically different ways. For example, + is generally used as an infix function symbol ("$a + b$"), whereas the notation $f(x)$ uses a function symbol f in prefix position with its argument in brackets. In Naproche, we distinguish six basic ways in which function symbols behave syntactically, and call these the *syntactic types* of the corresponding function symbols:

1. *infix*: Two-argument function symbol placed between its arguments (e.g. + in $n + m$).
2. *suffix*: One-argument function symbol placed after its argument (e.g. ! in $n!$).
3. *prefix*: One-argument function symbol placed before its argument (e.g. sin in $\sin x$).
4. *classical*: Function symbol with one or more arguments preceding its arguments, which are bracketed and separated by commas (e.g. f in $f(x)$ or $f(x, y)$).
5. *quantifier*: Two-place function symbol placed before its two arguments, where the arguments have to have types of the form $var(t_1, X)$ and $X - t_2$, and where the first argument position may be filled with a variable list rather than a single variable (e.g. $\forall x, y\ R(x, y)$).
6. *circumfix*: Expression for a function with one or more arguments, which are embedded into a predefined string of symbols, with at least one symbol at the beginning, at the end and between any two successive arguments (e.g. the degree of a field extension, $[K : k]$, considered as a two-place function depending on K and k). The *name* of a circumfix function is this predefined string with `[arg]` denoting the positions of its arguments. For example, the name of the field extension function is `[[arg]:[arg]]`.

[3] We use Prolog-like notation, i.e. capital letters for variables and $\_$ for an anonymous variable, when describing the type system.

Now consider an example from real analysis, namely the differentiation function, which is a function from differentiable real functions to real functions, sending any f to its derivative f'. When written in this $'$-notation, this function clearly has syntactic type *suffix*. But when we write $f'(x)$, we use the complex function name f' as a function with syntactic type *classical*. Now this does not seem to depend on the syntactic type of f: Suppose we have defined an extension of the factorial function ! to the reals (e.g. by $x! := \Gamma(x+1)$ using the Gamma function [12]). If we then apply its derivative $!'$ to some real x, we would write $!'(x)$ and not $x!'$[4]. So it seems to be inherent in the way the differentiation function symbol $'$ is used that the complex function name it produces is of syntactic type *classical*. We formalise this by saying that $'$ is of syntactic type *[suffix,classical]*. This means that its basic syntactic type is *suffix*, and the syntactic type of any function name whose head is $'$ is *classical*.

This machinery makes it possible to correctly handle many complicated notations: For example, exponentiation is treated as a function of syntactic type *[circumfix,suffix]* and of type $[i] => ([i] => i))$ (so in this case the notation we use makes us treat this multiple-argument function in the way such functions are treated in STT rather than using an inherent multiple-function type), where the name of the circumfix function is `^{[arg]}`. In the case of `x^{y}`, this function is first applied to `y`, yielding `^{y}`, which is considered a suffix function, so that applying it to x yields `x^{y}`.

In Naproche we distinguish two different kinds of math modes: The first is used for formulae (like $x = y^2$) and terms that serve as definite noun phrases (like $2x - 1$). The second is used for quantified terms, like the first two symbolic expressions in "For every x there is some $f(x)$ such that $R(x, f(x))$." Terms of the first kind are parsed by what we call the *normal formula grammar*, and terms of the second kind are parsed by what we call the *quantterm grammar*.

6 Normal Formula Grammar

Below we describe the normal formula grammar semi-formally by first listing (in a formal DCG-notation with Prolog-like syntax) a list of simplified grammar rules that any term must obey and then describing informally additional constraints that any term must satisfy in order to be actually parsed by the grammar. The constituent "term" used in the DCG rules below has an argument specifying the syntactic type of the term (i.e. a list of basic syntactic types). We use the variable name ST for a variable ranging over syntactic types.

Simplified normal formula grammar

term(ST) → term([classical|ST]), ['('], term_list, [')'].
term(ST) → term(_), term([suffix|ST]).

[4] Since this is a made-up example, we should add that our intuition as to what notation would be appropriate here has been confirmed by a number of mathematicians from the University of Bonn.

term(ST) → term([prefix|ST]), term(_).
term(ST) → term([quantifier|ST]), variable_list, term(_).
term(ST) → term(_), term([infix|ST]), term(_).
term(ST) → circumfix_term(ST).
term(ST) → ['('], term(ST), [')'].
term(ST) → variable(ST).

term_list → term(_), [','], term_list.
term_list → term(_).

variable_list → quantified_variable, [','], variable_list.
variable_list → quantified_variable.
quantified_variable → [_].
variable → [_].

For every predefined or accessible[5] *variable V of syntactic type ST, add a rule of the following form to the grammar:*
variable(ST) → V.

For every accessible circumfix function of syntactic type ST and name $S_1^1 \dots S_1^{n_1}[arg]S_2^1 \dots S_2^{n_2}[arg] \dots [arg]S_m^1 \dots S_m^{n_m}$*, add a rule of the following form to the grammar:*
circumfix_term(ST) →
$[S_1^1], \dots, [S_1^{n_1}], term(\_), [S_2^1], \dots, [S_2^{n_2}], term(\_), \dots, term(\_), [S_m^1], \dots, [S_m^{n_m}]$.

6.1 Operator Priorities

Syntactic disambiguation principles like the precedence of multiplication and division operators over addition and subtraction operators are encoded into the grammar using predefined operator priorities. We use the following operator priorities (in the order of decreasing precedence):

- +, −, → and ↔
- Prefix functions
- Suffix functions
- Other infix functions

Additionally, there is a principle which overrides the above operator priorities, namely that the operators used to form atomic formulae always have a higher precedence than the operators used to combine atomic formulae into complex formulae.

As an example for the functioning of these syntactic disambiguation principles,

[5] Given that Naproche's Proof Representation Structures are a variant of Discourse Representation Structures [14], "accessible" is to be understood as in Discourse Representation Theory. Basically, an accessible variable is a variable that was introduced in the preceding text that we can refer to by using the same variable name.

(1) $x + yz = \sin an! \wedge x = y \rightarrow z - y + z = 0$

is disambiguated as

(2) $(((x + (yz)) = \sin(a(n!))) \wedge (x = y)) \rightarrow (((z - y) + z) = 0)$.

In all cases that we are aware of, these syntactic disambiguation principles lead to an intuitive reading of the symbolic expression.

6.2 Defaultness of the Syntactic Type *Classical*

As already alluded in section 3, the syntactic type *classical* is the default syntactic type for newly introduced functions. This principle is implemented into the grammar by an additional constraint that in the second to fifth DCG rule specified above, as well as in the rule "variable → [_].", the syntactic type of a term may not be instantiated to *infix*, *prefix*, *quantifier*, *suffix* or *circumfix*. For example, the requirement of the final term to have "suffix" as syntactic type in the second rule means that this syntactic type must already be associated with the term when parsing it and may not be attached to the term afterwards. There is a limited list of predefined infix function symbol $(\cdot, +, -, *, ., \circ, /)$ for which this constraint does not apply.

In practice, this constraint means that when you are quantifying over a function, this function may be used with classical syntactic type or, if a preferred infix function symbol is used, with infix syntactic type, but not with prefix, suffix or quantifier syntactic type. So (3) and (4) are allowed, but (5), (6) and (7) (with z read as an infix, f as a prefix and g as a suffix function symbol) are not allowed.

(3) $\exists f\ f(a) = 0$

(4) $\exists *\ x * x = x$

(5) $\exists z\ xzx = x$

(6) $\exists f\ fa = 0$

(7) $\exists g\ ag = 0$

The defaultness of the syntactic type *classical* also explains why we don't formalise functions used in this syntactic way as circumfix functions. This would certainly be possible: A one-argument classical function f could also be considered a circumfix function with name `f([arg])`. However, this way we would not be able to account for the fact that a function that was introduced without fixing its syntactic type can be used with syntactic type *classical*.

6.3 Predefined Variables

It should be noted that we do not make the distinction between variables and constants that is usually made in the syntax of first-order logic and many other logical systems. In the semi-formal language of mathematics, there is a continuum between variable-like and constant-like expressions; this continuum is captured in Naproche through the use of dynamic quantification inherent in Discourse Representation Theory [14], so that the bivalent distinction used in first-order logic is not needed.

However, logical constants are still treated in a special way, namely as "predefined variables". These are also given a predefined type and syntactic type as follows:

Predefined variable	Type	Syntactic type
$\rightarrow$, $\leftrightarrow$, $\wedge$ and $\vee$	$[o, o] \rightarrow o$	*infix*
$\neg$	$[o] \rightarrow o$	*prefix*
$\forall$ and $\exists$	$[var(\_, X), X - o] \rightarrow o$	*quantifier*
$=$	$[T, T] \rightarrow o$[6]	*infix*
$\neq$	$[\_, \_] \rightarrow o$[7]	*infix*

6.4 Kinds of Variables

In the parsing process we distinguish different kinds of variables:

- Predefined variables (logical constants)
- Bound variables
- Variables that were implicitly introduced earlier on in the symbolic expression and are now reused
- Accessible variables whose antecedent is in the same sentence
- Accessible variables whose antecedent is in a preceding sentence
- Implicitly introduced variables

When trying to parse a variable, we always first try to parse it according to a variable kind higher up in the above list before trying the kinds lower down in the list. Once a variable has been parsed in one way, it may no longer be parsed in such a way as to be of a kind that is mentioned later in the above list than the kind that it has already been assigned. This means, for example, that if x is accessible and we parse $\exists x\ x + x = x$, then all instances of x in this formula are bound by the existential quantifier; none of the instances of x refers to the accessible variable.

6.5 Coverage of the Formula Grammar

The formula grammar can cope with almost all terms that serve as definite noun phrases and formulae found in mathematical texts. Here is a list of formulae that can be correctly parsed and disambiguated by it:

[6] i.e. the two arguments must be of the same type.
[7] i.e. the two arguments may be of distinct types.

$$x(y+z) = 0$$

$$x = y < z$$

$$x *_G x = x$$

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$$

$$x_0 \lim_{x \to x_0} f(x^2) = 2f(\frac{x_0^{x_0}}{2}) \neq f'(N!)$$

$$T = m_0 \frac{l^2}{2}((cos\varphi_0\varphi_0')^2 + (-sin\varphi_0\varphi_0')^2)$$

Of course, these formulae can only be parsed if the types and syntactic types of the function symbols appearing in them are known in advance. This information is created by the *quantterm* grammar described in section 7 when the functions are introduced.

There are some limitations of the current implementation of the formula grammar that we are aware of: Firstly our formula grammar can only handle variable binding if the occurrence of the variable that binds the other occurrences precedes the bound occurences. Hence the formula grammar cannot handle the integral notation of the form $\int f(x)dx$, where the first occurence of x is bound by the final occurence of x. Furthermore, the formula grammar can currently not cope with formula fragments like "$= 0$" nor with formulas containing triple dots like "$n \in \{1, \ldots, N\}$". However, we believe that the approach presented in this paper constitutes a framework for tackling even these harder cases, i.e. that the current limitations are not due to principle limitations of our approach, but rather due to the prototypical character of the implementation.

As already mentioned in the introduction, a quantitative evaluation of the coverage of the formula grammar is a highly nontrivial task. It involves reformulating the natural language context of the formulae in a controlled natural language, so that a full semantic analysis of the context can be achieved. This has so far only be accomplished for the first chapter of Landau's *Grundlagen der Analysis*, where the formula grammar parsed and correctly disambiguated all formulae [4].

7 Quantterm Grammar

Consider the following example text from [15]:

(8) Suppose that, for each vertex v of K, there is a vertex $g(v)$ of L such that $f(st_K(v)) \subset st_L(g(v))$. Then g is a simplicial map $V(K) \to V(L)$, and $|g| \backsimeq f$.

Here the natural language quantification "there is a vertex $g(v)$" locally introduces a new vertex to the discourse; but since the choice of the vertex depends on v and we are universally quantifying over v, it globally introduces a function

g to the discourse. In the next sentence there is an explicit reference to this implicitly introduced function.

Quantterms are symbolic expressions that appear in the scope of a natural language quantification, and are either just simple variables (in which case we call them *simple quantterms*), or, like in the above example, complex expressions that implicitly introduce a function to the discourse.

In order to discuss the functioning of quantterms, consider the following three example sentences:

(9) There is some y such that $R(y)$.

(10) For every x there is some y such that $R(x, y)$.

(11) For every x there is some $g(x)$ such that $R(x, g(x))$.

As described in [8], the premise added to the premise list for representing the information from (9) would not be $\exists y\ R(y)$, but $R(c_y)$ for a new constant symbol c_y. The reason for this replacement of existentially quantified variables by constant symbols is that the first-order quantifier $\exists$ does not have the dynamic properties of the natural language quantification with "there is": After stating sentence (9), we can later use the symbol y to refer to the same object that was introduced by this sentence. If we represented the content of the sentence by $\exists y\ R(y)$, then the scope of the y would only be this formula and could thus not include later uses of y. By using $R(c_y)$ for the content of (9) and replacing later uses of y by c_y, we do get the wanted coreference between the y in (9) and the later y.

This replacement of an existentially quantified variable by a constant is a special case of *skolemization* [2, 8]. In the representation of sentence (10) we make use of the more general kind of skolemization, which involves introducing new function symbols rather than new constant symbols. Its representation becomes $\forall x\ R(x, f_y(x))$, where f_y is a newly introduced function symbol; $f_y(x)$ replaces all occurrences of y in the scope of $\forall x$, where the argument x makes explicit that the choice of y depends on the value of x.

In the case of sentence (11), $g(x)$ can at first be considered to just be a complex variable name, usable in this very form later on in the sentence. Just as in the case of sentence (10), we skolemize this variable and make its dependencies explicit; in this case $g(x)$ depends on x. All this is the same as for sentence (10). But now we have to take into account that the author made this dependency explicit by writing $g(x)$ instead of y. This makes it possible to identify g with the skolem function that skolemization gives rise to, and to use this g as a function outside the universally quantified sentence in which $g(x)$ was introduced.

Now let us look at a somewhat more complex example:

(12) For all x, y there is some $g_x(y)$ such that $R(x, y, g_x(y))$.

After this sentence, we want to be able to use a function of syntactic type *[circumfix,classical]* named `g_{[arg]}`. So already when parsing the quantterm, we want to identify this syntactic type and name of the head function. This is

done by recursively allowing the head function of a quantterm to be again a quantterm. So in the case of $g(x)$ in (11), g may again be a quantterm, and is actually a simple quantterm. Now in the case of (12), g_x is first identified as head function of syntactic type *classical*, and is further analysed as quantterm. This further analysis recognises g_x as circumfix function `g_{[arg]}`.

7.1 Disambiguating Quantterms

Now one problem is that the quantterm grammar finds a number of possible readings for any input. For example, $f(x, y)$ can be interpreted in four ways:

1. as two-place classical function f (depending on x and y)
2. as two-place circumfix function `f([arg],[arg])` (depending on x and y)
3. as one-place circumfix function `f([arg],y)` (depending on x)
4. as one-place circumfix function `f(x,[arg])` (depending on y).

Here we want to choose the first reading as the preferred reading to be used by the program. This is done by a special algorithm for selecting the preferred reading, which works as follows:

- Non-*circumfix* readings are always preferred over *circumfix* readings.
- Between two *circumfix* readings, one is preferred over the other if its *circumfix* name has an `[arg]` at a place, where the other has a symbol.
- A reading that has *classical* in the second position of the syntactic type list is preferred over one that does not. (This principle is needed, for example, to ensure that in $f'(x)$, $'$ is interpreted as a suffix function making f' classical rather than as a classical function making $'(x)$ a suffix function.)
- When none of the above rules decides which reading is better, we recursively check which head function is preferred by those rules.

8 Disambiguation after Parsing

As mentioned in section 4, the type system is not capable of blocking all unwanted readings. This is due to the fact that our type system is not fine-grained enough. All objects that are not functions are of the same type, namely i. So, for example, both natural numbers and sets would be of the type i. If one has defined that for sets A, B, the expression A^B denotes the set of functions from A to B, and one has furthermore defined that for natural numbers m, n, the expression m^n denotes the nth power of m, then one has defined two functions of syntactic type *[circumfix,suffix]* and type $[i] \to ([i] \to i)$, both named `^{[arg]}`. Since their name, type and syntactic type are identical, they are indistinguishable during the parsing process. Thus, the ambiguity arising from this notational clash has to be resolved after the parsing process.

After updating the Proof Representation Structure with the representation of a parsed sentence, the Naproche system checks this added representation for logical correctness. This checking process involves the checking of presuppositions

[8]. The two just mentioned functions of equal name, type and syntactic type would trigger different presuppositions: The first would trigger the presupposition that both of its arguments are sets, whereas the second would trigger the presupposition that both of its arguments are numbers. Since it is not possible for both of these presuppositions to be fulfilled for a given pair of arguments, the ambiguity can certainly be removed in the process of checking the presuppositions[8].

It is also possible that the type information needed for disambiguating a symbolic expression is only available after the completion of the parsing process for that expression. Suppose, for example, that a user has defined a relation "$>$" on both natural numbers and functions of natural numbers, and uses the symbol 1 not only for the natural number 1, but also for the identity function. Now consider the following sentence:

(13) For all $x > 1$ such that $x^2 + 1$ is prime we have $R(x)$.

If the exponential notation x^2 is only defined for numbers and not for functions, then this sentence can be disambiguated using type information: x has to be of type i in "$x^2 + 1$" and therefore also in "$x > 1$", and so the "$>$" in "$x > 1$" refers to the relation on numbers and not the one on functions. But this type-based disambiguation of "$x > 1$" was not possible during the process of parsing "$x > 1$", because at that point "$x^2 + 1$" had not yet been parsed. In order to handle such type-based disambiguations that occur after that parsing of an expression, we use *type-dependency graphs*, which specify which reading of an expression depends on which type judgements. A detailed description of type-dependency graphs would, however, go beyond the scope of this paper.

9 Related Work

For understandable reasons, most formal mathematics systems simplify symbolic mathematics to a purely formal language, thus avoiding the issues that our paper is intended to tackle. Even languages of systems that clearly aim at a higher degree of naturality, like Mizar [17] and SAD [19], still largely treat the symbolic parts of mathematical texts like a formal language. The only work outside Naproche we are aware of that recognises the problem of parsing and disambiguating symbolic mathematics as intertwined with the natural language component of mathematical texts and as of a completely different kind than parsing formal languages is Ganesalingam [9]. He has analysed the language of mathematics – including symbolic mathematics – in much detail and developed a very ingenious theory for "a computer language which closely resembles the

[8] It is of course also possible that a user defines two clashing notations whose presuppositions may be fulfilled by the same argument(s); this, however, is almost certainly bad style, so that the user should get a warning from the system when this happens; nevertheless, the system does always choose one reading as the preferred one, using other heuristics, for example preferring notations that were defined later over ones that were defined earlier.

language used by human mathematicians in publications"[9]. We owe him a lot, since his work has enhanced our understanding of the language of mathematics and has helped us to develop the ideas presented in this paper. There are, however, two main differences between Ganesalingam's approach and ours:

Firstly, he has a methodological principle that no mathematical content is encoded directly into his theory, and he considers such syntactic disambiguation principles as the precedence of multiplication over addition as part of mathematical content[10]. Thus he does not encode such principles into his theory, but requires the author to write sentences of the following form in order to get the desired disambiguation of arithmetic expressions:

(14) If m, n and k are natural numbers, then "$m + nk$" means "$m + (nk)$".

We on the other hand do not want to require the author to write things that mathematicians do not normally write, and so decided to encode some basic syntactic disambiguation principles directly into our theory.

Secondly, as already alluded in section 4, he relies much more heavily on a type system than we do for disambiguating symbolic mathematics. This is due to the fact that he does not include presuppositions into the disambiguation machinery. By making use of presuppositions for disambiguation, we were able to attain similar goals as Ganesalingam with a much more coarse type system. One of the benefits of the coarseness of the type system is that we do not require the author to make statements whose only goal is to influence the typing of symbolic material.

10 Conclusion

We have presented the difficulties that a computer program for analysing mathematical texts faces with respect to symbolic mathematics, given that the input language is to be as similar as possible to the language that mathematicians commonly use in journals and textbooks. We have described how these difficulties are solved in the Naproche system, and compared this solution to Ganesalingam's solution.

References

1. Avigad, J., Donnelly, K., Gray, D., Raff, P.: A formally verified proof of the prime number theorem. ACM Transactions on Computational Logic 9(1:2) (2007)
2. Brachman, R., Levesque, H.: Knowledge Representation and Reasoning. Morgan Kaufmann Publishers, Massachusetts (2004)
3. de Bruijn, R.G.: Reflections on Automath. In: Nederpelt, R.P., et al. (eds.) Selected Papers on Automath. Studies in Logic, vol. 133, pp. 201–228, 215. Elsevier, Amsterdam (1994)

[9] Page 9 in [9]
[10] Page 105 in [9]).

4. Carl, M., Cramer, M., Kühlwein, D.: Chapter 1 from Landau in Naproche 0.5 (2011), http://naproche.net/downloads/2011/landauChapter1.pdf
5. Church, A.: A Formulation of the Simple Theory of Types. The Journal of Symbolic Logic 5(2), 56–68 (1940)
6. Cramer, M., Koepke, P., Kühlwein, D., Schröder, B.: The Naproche System. Calculemus, Emerging Trend Paper (2009)
7. Cramer, M., Fisseni, B., Koepke, P., Kühlwein, D., Schröder, B., Veldman, J.: The Naproche Project – Controlled Natural Language Proof Checking of Mathematical Texts. In: Fuchs, N.E. (ed.) CNL 2009. LNCS, vol. 5972, pp. 170–186. Springer, Heidelberg (2010)
8. Cramer, M., Kühlwein, D., Schröder, B.: Presupposition Projection and Accommodation in Mathematical Texts. In: Semantic Approaches in Natural Language Processing: Proceedings of the Conference on Natural Language Processing 2010 (KONVENS), pp. 29–36. Universaar (2010)
9. Ganesalingam, M.: The Language of Mathematics, PhD thesis, University of Cambridge (2009)
10. Hales, T.: Jordan's proof of the Jordan Curve theorem. Studies in Logic, Grammar and Rhetoric 10(23) (2007)
11. Hales, T.: Introduction to the Flyspeck Project. Dagstuhl Seminar Proceedings (2006)
12. Heuser, H.: Lehrbuch der Analysis. In: Teil 2, 6th edn., B.G. Teubner, Stuttgart (1991)
13. Kadmon, N.: Formal Pragmatics. Wiley-Blackwell, Oxford, UK (2001)
14. Kamp, H., Reyle, U.: From Discourse to Logic: Introduction to Model-theoretic Semantics of Natural Language. Kluwer Academic Publishers, Dordrecht (1993)
15. Lackenby, M.: Topology and Groups. Lecture Notes (2008), http://people.maths.ox.ac.uk/lackenby/tg050908.pdf
16. Landau, E.: Grundlagen der Analysis, 3rd edn (1960)
17. Matuszewski, R., Rudnicki, P.: Mizar: The first 30 years. Mechanized Mathematics and Its Applications 4 (2005)
18. Ranta, A.: Structure grammaticales dans le français mathématique II (suite et fin). Mathématiques, Informatique et Sciences Humaines 139, 5–36 (1997)
19. Verchinine, K., Lyaletski, A., Paskevich, A.: System for Automated Deduction (SAD): a tool for proof verification. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 398–403. Springer, Heidelberg (2007)

Interleaving Strategies

Bastiaan Heeren[1] and Johan Jeuring[1,2]

[1] School of Computer Science, Open Universiteit Nederland
P.O. Box 2960, 6401 DL Heerlen, The Netherlands
{bhr,jje}@ounl
[2] Department of Information and Computing Sciences, Universiteit Utrecht

Abstract. Rewrite strategies are used to specify how mathematical exercises are solved in interactive learning environments, and to provide feedback to students solving such exercises. We have developed a generic strategy language with which we can specify rewrite strategies in many (mathematical) domains. Although our strategy language is quite powerful, it lacks an essential component for specifying strategies, namely the *interleaving* of two strategies. Often students have to perform multiple subtasks, but the order in which these tasks are performed is irrelevant, and steps of solutions may be interleaved. We show the need for combinators that support interleaving by means of several examples. We extend our strategy language with different combinators for interleaving, define the semantics of the extension, and show how the interleaving combinators are implemented in the parsing framework we use for recognizing student behavior and providing hints.

Keywords: strategy language, interleaving, feedback.

1 Introduction

Strategies specify how a wide range of exercises can be solved incrementally, such as bringing a logic proposition to disjunctive normal form, reducing a matrix, solving a quadratic equation, or calculating with fractions. In previous work [13] we have developed a language for rewrite strategies for exercises and a framework for feedback services built on top of this language, which is now used in intelligent tutoring systems such as MathDox [7], the Digital Mathematics Environment of the Freudenthal Institute [8], and the ActiveMath system [17] to follow student behavior, report applications of buggy rules, give hints, and show worked-out examples.

Rewrite strategies for exercises and the strategy language in which they are formulated should satisfy a number of requirements:

- The strategy language is *generic*: it can be used for any domain in which exercises are solved by incrementally applying rules, such as logic, algebra, programming, etc.
- It is possible to *automatically calculate feedback* given a strategy and actions of a user on an exercise that is solved using the strategy.

J.H. Davenport et al. (Eds.): Calculemus/MKM 2011, LNAI 6824, pp. 196–211, 2011.

- Strategies satisfy the *cognitive fidelity* principle [3]: they reflect textbook descriptions of procedures for solving exercises.
- Strategies are *observable*: we can print, inspect, adapt, and even transform strategies, so that teachers can use variants of a strategy, and students can customize the level of feedback when solving an exercise [12].
- Strategies are *compositional*: a strategy can be reused verbatim in another strategy.

Our strategy language satisfies these requirements to a large extent, but we have encountered a number of cases where some of the above requirements are not fulfilled. These cases are related to the cognitive fidelity principle and the compositionality requirement. For example, when solving the equation $x^2(2x^2 - 1) = 4(2x^2 - 1)$, the textbook procedure says: apply the rule $AC{=}BC \Rightarrow A{=}B \vee C{=}0$ to obtain the two quadratic equations $x^2 = 4$ and $2x^2 - 1 = 0$, and then solve these equations. So clearly the strategy for solving quadratic equations is reused in the strategy for solving equations of a higher-degree. When presenting a solution to a student, we want to first solve one quadratic equation, and then the other. But a student may solve the two equations in any order, and even switch halfway from one equation to the other. At the moment it is very hard to satisfy both requirements: our strategy for higher-degree equations does reuse the quadratic strategy, but it does not satisfy the cognitive fidelity principle. The main cause for this is the fact that we lack a language component for expressing that two strategies have to be solved, but the order in which they are solved does not matter, and steps for solving the strategies may be interleaved. For such functionality, we need an *interleaving* combinator for strategies.

Interleaving is a common operator in communicating sequential processes (CSP) [14]. It also appears under the names parallel and merge [4], but we prefer the name interleave, because it best describes the semantics of the operator we need. Parallel normally suggests that actions are performed simultaneously, which is not important in our case. In this paper we show how to extend our strategy language with several constructs to interleave strategies. The main interleave strategy combinator takes two strategies as argument, and allows a student to take steps from either of the two strategies, and finishes whenever both argument strategies are finished. We describe the semantics of the added constructs, derive properties for them, and show how they are implemented in our framework. The implementation is rather challenging because of the presence of so-called "administrative rules", which are rules that are silently applied (e.g., for navigating through a term). These rules are essential for our framework, but are not directly derived from user actions. With the extended strategy language we can more easily compose strategies, write strategies in a more natural fashion, and provide better feedback to students. The contributions of this paper are an explanation of the importance of adding facilities for specifying interleaving to a rewrite strategy language for exercises, and an implementation of the extended strategy language as a parser that recognizes student behavior and gives feedback.

This paper is organized as follows. In Section 2 we explain the need for interleaved strategies by means of several examples. Section 3 adds an interleaving strategy combinator to our strategy language, and gives its semantics. Section 4 shows how to implement an interleaving combinator for strategies in our framework, after which Section 5 discusses several design decisions about dealing with administrative rules. Section 6 shows how the added combinators help in formulating strategies for our examples. Section 7 discusses related and future work, and concludes.

2 Examples

For many exercises it is essential that we can specify that two (or more) strategies should be interleaved. This section gives examples, and discusses why existing strategy language constructs are not sufficiently expressive for these exercises.

Example: applying tautologies. In many courses on logic students are asked to rewrite logic expressions to some normal form, such as the disjunctive normal form (DNF). There are several procedures for rewriting a logic expression to DNF: one is to propagate all occurrences of the constants *true* and *false*, then replace implications and equivalences by their definitions, push negations top-down inside to the leaves of the logic expression using the rules for negation, and, finally, distribute $\wedge$ over $\vee$ to reach DNF. Furthermore, whenever a tautology or contradiction appears, simplify the formula by turning it into a constant. The first four steps are nicely captured in a strategy, but we need special machinery to model the last step for tautologies and contradictions. We could replace every rule that appears in the first four steps by the choice of that rule and the rules for tautologies and contradictions. However, such a transformation is not compositional, and bloats up strategies into huge artifacts that are hard to understand and maintain.

Example: solving polynomial equations of higher-degree. Suppose a student solves higher-degree equations in an interactive learning environment. Some higher-degree equations can be solved by applying the rule $AC{=}BC \Rightarrow A{=}B \vee C{=}0$, where the equations in the right-hand side of the rule have at most degree two. Of course, it does not matter in which order the student solves the resulting equations $A = B$ and $C = 0$, and she might even switch between solving the equations halfway. However, when we present a solution to a student, we prefer not to switch between solving the two equations. How do we describe a strategy that accepts this student behavior, and still gives hints to the student when she asks for it? We can easily express that a student should first solve the first equation and then the second, or vice versa, but this disallows switching between the two equations halfway. At the moment, we use a flexible strategy to solve these equations, which expresses that any rule from the quadratic strategy can be applied anywhere in the two equations. Since this leads to many choices, we have specified an order on the rules, and preferred rules are applied first. As a

consequence, our worked-out solution interleaves steps in solving the two equations, but sometimes in a non-intuitive way. The strategy violates the cognitive fidelity principle because not all solutions reflect the textbook description. The fact that we use a flexible strategy which accepts any rule from a particular set of rules violates the compositionality requirement: although we instruct students to apply the rule $AC{=}BC \Rightarrow A{=}B \vee C{=}0$ and then solve the resulting quadratic equations, we do not reuse the strategy for quadratic equations because that would disallow some student behavior. In this case, it was possible to relax the strategy for solving quadratic equations, such that students are allowed to switch between multiple equations that come from a single higher-degree equation. In more complicated situations, this approach might no longer be feasible.

More examples. We have used our strategy framework to develop an intelligent tutoring system for learning functional programming [10]. The tool supports the gradual refinement of programs until a program is determined to be equivalent to a model solution. Programs under development may contain one or more "holes" that need to be further refined. A student can refine these holes in any order, and a refinement step can introduce new holes. Again, to follow and support this behavior, we need a way to specify that an exercise consists of a number of strategies that can be solved in an interleaving fashion.

Other example applications which would profit from an interleaving construct for strategies are found in specifying rules for cleaning up expressions after a rewrite rule has been applied, and in tools for teaching theorem proving [16]. If such a tool has to follow the actions of a student and allow her to work on any of the subtrees to be proven, then interleaved strategies are needed for exactly the same reasons as for the functional programming tutor. Interleaving also solves the simpler problem of specifying a strategy that requires a number of rules to be applied once, but the order is irrelevant. At the moment we specify this by repeating the rules until they cannot be applied anymore, which often amounts to the same thing, but which will not work in more complicated situations.

Until now we have developed our strategies without interleaving constructs for strategies. This has led to strategies that are less precise or too strict, and harder to maintain, reuse, and adapt. Sometimes, this leads to problems in using our tools. For example, the "applying tautologies" example described above appears at the top of the list of suggested improvements to our tool for rewriting logic expressions to DNF. It is possible to specify the interleaving of strategies explicitly, but this would give huge strategies: the text size of explicitly specifying the interleaving of two strategies is more than exponential in the text size of the two argument strategies. Concluding, we need to add an interleaving strategy combinator to our strategy language.

3 Interleaving and Rewrite Strategies

In this section we show how to add interleaving to our strategy language. We will first explore the concepts of interleaving and atomicity, after which we extend

the strategy language. The notation we adopt, for interleaving and for rewrite strategies, is inspired by the algebra of communicating processes (ACP) [4]. The concise syntax makes it suitable for the type of specifications found in this paper (compared to the implementation-oriented syntax used in other papers [13,12]). Although we use a mathematical notation in the rest of this paper, the definitions directly correspond to programs in a functional programming language like Haskell [19], and we use Haskell's semantics for recursive equations defining functions.

3.1 Interleaving Sentences

We use $a, b, c, ...$ to denote symbols, and x, y, z for sentences (sequences) of such symbols. As usual, we write ϵ for the empty sequence, and xy (or ax) for concatenation. We start by defining the interleaving of two sentences ($x \parallel y$): this operator can be defined conveniently in terms of left-interleave (denoted by $x \lfloor\!\lfloor y$, and also known as the left-merge operator [4]), which expresses that the first symbol should be taken from the left-hand side operand. ACP traditionally defines interleave in terms of left-interleave (and "communication interleave") to obtain a sound and complete axiomatization for the algebra of communicating processes [9].

$$\begin{array}{lcl} \epsilon \parallel x & = & \{x\} \\ x \parallel \epsilon & = & \{x\} \\ x \parallel y & = & x \lfloor\!\lfloor y \cup y \lfloor\!\lfloor x \quad (x \neq \epsilon \wedge y \neq \epsilon) \end{array} \qquad \begin{array}{lcl} \epsilon \lfloor\!\lfloor y & = & \emptyset \\ ax \lfloor\!\lfloor y & = & \{az \mid z \in x \parallel y\} \end{array}$$

For example, the result of interleaving the sentences abc and de (that is, $abc \parallel de$) results in the following set:

$$\{abcde, abdce, abdec, adbce, adbec, adebc, dabce, dabec, daebc, deabc\}$$

The set $abc \lfloor\!\lfloor de$ only contains the six sentences that start with symbol a. It is worth noting that the number of interleavings for two sentences of lengths n and m equals $\frac{(n+m)!}{n!m!}$. This number grows quickly with longer sentences. An alternative definition of interleaving two sequences, presented by Hoare in his influential book on CSP [14], is by means of three laws:

$$\begin{array}{rcl} \epsilon \in (y \parallel z) & \Leftrightarrow & y = z = \epsilon \\ x \in (y \parallel z) & \Leftrightarrow & x \in (z \parallel y) \\ ax \in (y \parallel z) & \Leftrightarrow & (\exists\, y' : y = ay' \wedge x \in (y' \parallel z)) \\ & \vee & (\exists\, z' : z = az' \wedge x \in (y \parallel z')) \end{array}$$

3.2 Interleaving Sets

The operations for interleaving sentences can be lifted to work on sets of sentences by considering all combinations of elements from the two sets. Let X, Y, and Z be sets of sentences. The lifted operators are defined as follows:

$$\begin{array}{lcl} X \parallel Y & = & \bigcup \{x \parallel y \mid x \in X, y \in Y\} \\ X \lfloor\!\lfloor Y & = & \bigcup \{x \lfloor\!\lfloor y \mid x \in X, y \in Y\} \end{array}$$

For instance, $\{a, ab\} \parallel \{c, cd\}$ yields a set containing 14 elements:

$$\{abc, abcd, ac, acb, acbd, acd, acdb, ca, cab, cabd, cad, cadb, cda, cdab\}$$

From these definitions, it follows that the lifted operator for interleaving is commutative, associative, and has $\{\epsilon\}$ as identity element. The left-interleave operator is not commutative nor associative, but has the interesting property that $(X \between Y) \between Z$ is equal to $X \between (Y \parallel Z)$.

3.3 Atomicity

In the case of rewrite strategies, it is useful to have a notion of atomic blocks within sentences. In such a block, no interleaving should occur with other sentences. We write $\langle x \rangle$ to make sequence x atomic: if x is a singleton, the angle brackets may be dropped. Atomicity obeys some simple laws:

$$\begin{array}{rcll} \langle \epsilon \rangle & = & \epsilon & \text{(the empty sequence is atomic)} \\ \langle a \rangle & = & a & \text{(all primitive symbols are atomic)} \\ \langle x \langle y \rangle z \rangle & = & \langle xyz \rangle & \text{(nesting of atomic blocks has no effect)} \end{array}$$

In particular, it follows that $\langle \langle x \rangle \rangle = \langle x \rangle$. Atomic blocks nicely work together with the definitions given for the interleaving operators, including the lifted operators: sentences now consist of a sequence of atomic blocks, where each block itself is a non-empty sequence of symbols. For instance, $a\langle bc \rangle \parallel \langle de \rangle f$ will return:

$$\{a\langle bc\rangle\langle de\rangle f, a\langle de\rangle\langle bc\rangle f, a\langle de\rangle f\langle bc\rangle, \langle de\rangle a\langle bc\rangle f, \langle de\rangle af\langle bc\rangle, \langle de\rangle fa\langle bc\rangle\}$$

In the end, when no more interleaving takes place, the blocks have no longer any meaning, and can be discarded.

Permuting sentences (i.e., enumerating all different orderings of a list of sentences, and concatenating these sentences) can be thought of as a simpler form of interleaving. More specifically, the sentences themselves should not be interleaved, which can be done by making the sentences atomic. Hence, we define *permute* $[x_1, ..., x_n]$ as $\langle x_1 \rangle \parallel ... \parallel \langle x_n \rangle$. For example,

$$\mathit{permute}\ [ab, cde, f] = \{\, abcdef, abfcde, cdeabf, cdefab, fabcde, fcdeab \,\}$$

3.4 Interleaving Strategies

A rewrite strategy is a context free grammar with rewrite rules as terminal symbols. A rewrite strategy is defined in terms of *strategy combinators*, and is described by the following grammar:

$$\sigma \ ::= \ 0 \mid 1 \mid r \mid \sigma + \sigma \mid \sigma \cdot \sigma \mid \mu\, f_\sigma \mid \ell\, \sigma$$

The basic components (symbols) of our language are rewrite rules r. Two (sub)-strategies can be combined into a strategy using the choice $(+)$ or sequence $(\cdot)$

combinator, with 0 (always fails) and 1 (always succeeds) as its unit element, respectively. The main purpose of our strategy language is to track student behavior, and to automatically calculate feedback based on the strategy and the current term. For this purpose we need to mark positions in the strategy, for which we use labels (ℓ). Such a label can, for example, be associated with a feedback text related to its particular position in the strategy.

Strategies can have recursive parts, at arbitrary positions. We use the fixpoint operator $\mu\ f_\sigma = f_\sigma\ (\mu\ f_\sigma)$ for this, where f_σ is a function that takes a strategy and returns one. With this operator, numerous derived combinators can be added to the strategy language, such as $many\ \sigma = \mu x.1 + \sigma \cdot x$.

The *language* (or semantics) of a strategy is a set of sentences, where each sentence is a sequence of (atomic blocks of) rewrite rules. Function $\mathcal{L}$ generates the language of a strategy, by interpreting it as a context-free grammar.

$$
\begin{array}{lcllcl}
\mathcal{L}\ (0) & = & \emptyset & \mathcal{L}\ (\sigma_1 + \sigma_2) & = & \mathcal{L}\ (\sigma_1) \cup \mathcal{L}\ (\sigma_2) \\
\mathcal{L}\ (1) & = & \{\epsilon\} & \mathcal{L}\ (\sigma_1 \cdot \sigma_2) & = & \{xy \mid x \in \mathcal{L}\ (\sigma_1), y \in \mathcal{L}\ (\sigma_2)\} \\
\mathcal{L}\ (r) & = & \{r\} & \mathcal{L}\ (\mu\ f_\sigma) & = & \mathcal{L}\ (f_\sigma\ (\mu\ f_\sigma)) \\
 & & & \mathcal{L}\ (\ell\ \sigma) & = & \{\textsc{Enter}_{(\ell)}\ x\ \textsc{Exit}_{(\ell)} \mid x \in \mathcal{L}\ (\sigma)\}
\end{array}
$$

With this semantics, it is easy to verify that the combinators $(+)$ and $(\cdot)$ form a semiring, as one would expect. This interpretation introduces the special rules Enter and Exit (parameterized by some label ℓ) that show up in sentences. These rules are used to trace positions in strategies. In Section 5 we discuss the subtleties of labels in strategies.

We extend the strategy language with new constructs for atomicity, interleaving, and left-interleaving:

$$\sigma\ ::=\ \ldots\ \mid\ \langle\sigma\rangle\ \mid\ \sigma \parallel \sigma\ \mid\ \sigma \between \sigma$$

The semantics for the new constructs is defined in terms of the lifted operators:

$$
\begin{array}{lcl}
\mathcal{L}\ (\langle\sigma\rangle) & = & \{\langle x\rangle \mid x \in \mathcal{L}\ (\sigma)\} \\
\mathcal{L}\ (\sigma_1 \parallel \sigma_2) & = & \mathcal{L}\ (\sigma_1) \parallel \mathcal{L}\ (\sigma_2) \\
\mathcal{L}\ (\sigma_1 \between \sigma_2) & = & \mathcal{L}\ (\sigma_1) \between \mathcal{L}\ (\sigma_2)
\end{array}
$$

Of course, more variations of interleaving can be added to the strategy language in a similar fashion, such as a combinator for permuting strategies. A second example is a variant of interleave that always takes steps from the left-hand side strategy if this is possible (and only if this fails, steps from the right operand), and finishes when no more steps can be done on either side.

The interleaving strategy combinator inherits the properties of the lifted interleaving operator that works on sets: it is commutative and associative, and has 1 as identity element. Because interleaving distributes over choice (that is, $\sigma_1 \parallel (\sigma_2 + \sigma_3) = (\sigma_1 \parallel \sigma_2) + (\sigma_1 \parallel \sigma_3)$), we have a second semiring. Also left-interleave distributes over choice. The operator that makes a strategy atomic is idempotent, and distributes over choice $\langle\sigma_1 + \sigma_2\rangle = \langle\sigma_1\rangle + \langle\sigma_2\rangle$. Many more properties can be found in the literature on ACP [4]. We use the properties of the strategy combinators for several purposes:

- Our implementation can be tested against these properties, and we have done so using the QuickCheck tool [6].
- The properties help strategy writers to reason about their strategies, and it provides insight into how the combinators behave.
- They will prove useful in defining the strategy recognizer, which is the topic of the next section.

4 Implementing Interleaving Strategies

Section 3 defines a language to specify rewrite strategies for exercises, extended with interleaving combinators. The definition of the semantics of this language is not suitable for implementing feedback services such as following the behavior of students, and giving hints and worked-out examples. For this, we need to develop a parser that can recognize student actions, give the next expected symbol when a student asks for a hint, or generate a complete worked-out example.

For recognizing sentences, it is sufficient to define the functions *empty* and *firsts* [13]. With these functions, input symbols can be consumed one after another, from left to right. Before we discuss how to implement the functions for the extended strategy language, we first have a look at an alternative specification for the interleaving combinator from an "operational" perspective.

We have three scenarios for parsing the strategy $\sigma_1 \parallel \sigma_2$: start with input for σ_1 (represented by $\sigma_1 \between \sigma_2$), start with σ_2, or test for the empty sentence.

$$\mathcal{L}\ (\sigma_1 \parallel \sigma_2)\ =\ \mathcal{L}\ (\sigma_1 \between \sigma_2) \cup \mathcal{L}\ (\sigma_2 \between \sigma_1) \cup \{\epsilon \mid \epsilon \in \mathcal{L}\ (\sigma_1) \cap \mathcal{L}\ (\sigma_2)\}$$

In this definition interleaving stops only when both strategies have the empty sentence, which is what the first law in Hoare's definition expresses.

4.1 Defining *Empty*

The function *empty* tests whether or not the empty sentence is generated by a strategy: $empty\ (\sigma) = \epsilon \in \mathcal{L}\ (\sigma)$. The direct translation of this specification of *empty* to a functional program, using the definition of language $\mathcal{L}$, gives a very inefficient program. Instead, we derive the following recursive function from this characterization, by performing case analysis on strategies:

$$\begin{array}{lcllcl}
empty\ (0) & = & false & empty\ (\sigma_1 + \sigma_2) & = & empty\ \sigma_1 \vee empty\ \sigma_2 \\
empty\ (1) & = & true & empty\ (\sigma_1 \cdot \sigma_2) & = & empty\ \sigma_1 \wedge empty\ \sigma_2 \\
empty\ (r) & = & false & empty\ (\langle\sigma\rangle) & = & empty\ \sigma \\
empty\ (\mu\ f_\sigma) & = & empty\ (f_\sigma\ 0) & empty\ (\sigma_1 \parallel \sigma_2) & = & empty\ \sigma_1 \wedge empty\ \sigma_2 \\
empty\ (\ell\ \sigma) & = & false & empty\ (\sigma_1 \between \sigma_2) & = & false
\end{array}$$

These equations follow almost directly from the specification of $\mathcal{L}$. There is no need to visit the recursive parts to determine the *empty* property for a strategy. The definition makes explicit that the left-interleave combinator never yields the empty sentence. The new definition for $\mathcal{L}\ (\sigma_1 \parallel \sigma_2)$ shows that both σ_1 and σ_2 need to have the empty property, otherwise $\epsilon \notin \mathcal{L}\ (\sigma_1 \parallel \sigma_2)$. Interpreting these equations for *empty* as a Haskell program gives an efficient program that is linear in the size of the argument strategy.

4.2 Defining *Firsts*

Given some strategy σ, the function *firsts* returns every rule that can start a sentence for σ, paired with a strategy that represents the remainder of that sentence. This is made more precise in the following specification (where r represents a rule, and x a sequence of rules):

$$\forall r, x : \; rx \in \mathcal{L}\ (\sigma) \;\Leftrightarrow\; \exists \sigma' : (r, \sigma') \in \mathit{firsts}\ (\sigma) \wedge x \in \mathcal{L}\ (\sigma')$$

As for the function *empty*, the direct translation of this specification into a functional program is infeasible. We derive an efficient implementation for *firsts* by performing a case analysis on strategies. The *firsts* set for the left-interleave case is somewhat challenging: this is exactly where we must deal with interleaving and atomicity. For a strategy $\sigma_1 \between \sigma_2$, we split σ_1 into an atomic part and a remainder, i.e., $\langle \sigma_1' \rangle \cdot \sigma_1''$. After σ_1' without the empty sentence, we can continue with $\sigma_1'' \between \sigma_2$. This approach is summarized by the following property, where the use of rule r takes care of the non-empty condition:

$$(\langle r \cdot \sigma_1 \rangle \cdot \sigma_2) \between \sigma_3 \;=\; \langle r \cdot \sigma_1 \rangle \cdot (\sigma_2 \between \sigma_3)$$

Function *split* transforms a strategy into alternatives of the form $\langle r \cdot \sigma_1 \rangle \cdot \sigma_2$:

$$\begin{array}{lcl}
\mathit{split}\ (0) & = & \emptyset \\
\mathit{split}\ (1) & = & \emptyset \\
\mathit{split}\ (r) & = & \{\, \langle r \cdot 1 \rangle \cdot 1 \,\} \\
\mathit{split}\ (\sigma_1 + \sigma_2) & = & \mathit{split}\ \sigma_1 \cup \mathit{split}\ \sigma_2 \\
\mathit{split}\ (\sigma_1 \cdot \sigma_2) & = & \{\, \langle r \cdot x \rangle \cdot (y \cdot \sigma_2) \mid \langle r \cdot x \rangle \cdot y \in \mathit{split}\ \sigma_1 \,\} \\
 & \cup & \textbf{if}\ \mathit{empty}\ \sigma_1\ \textbf{then}\ \mathit{split}\ \sigma_2\ \textbf{else}\ \emptyset \\
\mathit{split}\ (\mu\ f_\sigma) & = & \mathit{split}\ (f_\sigma\ (\mu\ f_\sigma)) \\
\mathit{split}\ (\ell\ \sigma) & = & \mathit{split}\ (\textsc{Enter}_{(\ell)} \cdot \sigma \cdot \textsc{Exit}_{(\ell)}) \\
\mathit{split}\ (\langle \sigma \rangle) & = & \{\, \langle r \cdot (x \cdot y) \rangle \cdot 1 \mid \langle r \cdot x \rangle \cdot y \in \mathit{split}\ \sigma \,\} \\
\mathit{split}\ (\sigma_1 \between \sigma_2) & = & \mathit{split}\ (\sigma_1 \between\!\!\!\between \sigma_2) \cup \mathit{split}\ (\sigma_2 \between\!\!\!\between \sigma_1) \\
\mathit{split}\ (\sigma_1 \between\!\!\!\between \sigma_2) & = & \{\, \langle r \cdot x \rangle \cdot (y \between \sigma_2) \mid \langle r \cdot x \rangle \cdot y \in \mathit{split}\ \sigma_1 \,\}
\end{array}$$

We briefly discuss the definitions for the new constructs:

- *Case* $\langle \sigma \rangle$: because atomicity distributes over choice, we can consider the elements of *split* σ (the recursive call) one by one. The transformation $\langle \langle r \cdot x \rangle \cdot y = \langle r \cdot (x \cdot y) \rangle \cdot 1$ is proven by first removing the inner atomic block, and basic properties of sequence.
- *Case* $\sigma_1 \between \sigma_2$: expressing this strategy in terms of left-interleave is justified by the definition of $\mathcal{L}\ (\sigma_1 \between \sigma_2)$ given in this section. For function *split*, we only have to consider the non-empty sentences.
- *Case* $\sigma_1 \between\!\!\!\between \sigma_2$: left-interleave can be distributed over the alternatives. Furthermore, $(\langle r \cdot x \rangle \cdot y) \between\!\!\!\between \sigma_2 = \langle r \cdot x \rangle \cdot (y \between \sigma_2)$ follows from the definition of left-interleave on sentences (with atomic blocks).

With the function *split*, we can now define the function *firsts* needed for most of our feedback services:

$$\mathit{firsts}\ (\sigma) \;=\; \{\, (r, x \cdot y) \mid \langle r \cdot x \rangle \cdot y \in \mathit{split}\ \sigma \,\}$$

5 Dealing with Administrative Rules

Our strategy framework uses *administrative rules* to change the context of an expression, but not the expression itself. Students cannot observe the application of administrative rules. Examples of such rules are tracking the labels in a strategy, keeping a focus (on a subexpression), and reading (or writing) a value from an environment. This section describes practical issues with administrative rules that arise when adding interleaving to the strategy language. We discuss how to deal with labels and navigation actions for moving the point in focus.

5.1 Labels in Strategies

Consider the sentences generated by the following strategy:

$$\ell_1\ (r_1 \cdot \ell_2\ (r_2 \cdot r_3) \parallel \ell_3\ (r_4 \cdot r_5))$$

When ignoring labels, this strategy generates 10 ($= \frac{5!}{2!3!}$) sentences. For each labeled (sub)strategy, we insert administrative rules to mark where we enter or leave that strategy: $\ell\ \sigma$ thus becomes $\textsc{Enter}_{(\ell)} \cdot \sigma \cdot \textsc{Exit}_{(\ell)}$. These markings tell the framework where a student is in a strategy, and allows us to give appropriate feedback based on the labels. However, the extra steps also significantly increase the number of sentences ($\frac{11!}{4!7!} = 330$). This explosion in the number of sentences quickly makes the strategy unusable for the purpose of generating feedback and tracking student behavior: there are too many possibilities to choose from.

Since users cannot observe administrative rules, the sentences with the administrative rules do not add interleavings that are interesting for a user. It does not make sense to switch to another interleaved strategy right after an enter or exit step, and before a major (non-administrative) step is detected. Hence, we allow the prefixes $\textsc{Enter}_{(\ell_1)} r_1$ and $\textsc{Enter}_{(\ell_3)} r_4$, but we disallow $\textsc{Enter}_{(\ell_1)} \textsc{Enter}_{(\ell_3)}$ and $\textsc{Enter}_{(\ell_3)} \textsc{Enter}_{(\ell_1)}$. This gives us again 10 sentences.

As a consequence of the administrative rules, we need variants of *empty* and *firsts* that skip over these rules and behave properly in the presence of interleaved parts, along the lines of the big step operator defined by Gerdes et al. [11]. For administrative rule r we have that $r \cdot \langle\sigma\rangle$ can be transformed into $\langle r \cdot \sigma\rangle$. This property can be used to refine the *split* function for the left-interleave case.

5.2 Navigation Actions

Many rewrite strategies in mathematics rely on navigation combinators that move the focus to a particular subexpression. For this, we use the zipper datastructure [15], and its operations such as moving up and down in a tree. In our strategies, we use the administrative rules Up and Downs. The rule for moving downwards returns multiple (zero or more) alternatives with a new focus, one for each child. With these navigation rules, we can define the *somewhere* combinator:

$$\mathit{somewhere}\ \sigma = \mu x.\sigma + (\textsc{Downs} \cdot x \cdot \textsc{Up})$$

Again, interleaving poses an additional challenge when dealing with administrative rules, this time for navigation. Consider the strategy $(\mathit{somewhere}\ \sigma_1) \parallel \sigma_2$, and assume that the rules in σ_2 take the current focus into account. It is undesirable to interleave the navigation actions from the left-hand side strategy with σ_2. This gives highly unpredictable behavior, especially when σ_2 also performs navigation actions. Therefore, we make the result of *somewhere* atomic by default. When moving the focus down, all interleaved strategies are blocked until the matching up action is recognized. We do the same for the other navigation combinators (e.g., *topDown*, which applies a strategy at the highest possible position in a tree). Note that a *somewhere* combinator that does allow interleaving (for instance, because the other strategy is known to ignore the focus) can still be defined if desired.

Besides navigation, there are other ways in which rules of interleaved strategies can interfere. Examples are rules that read and write values to an environment, or a rule that assumes a certain invariant to hold when it is executed. For such cases, the strategy developer has to make parts of the strategy atomic, or take other measures to ensure non-interference. This sometimes makes developing strategies significantly more complex, which is not uncommon when concurrency is involved.

6 Examples Revisited

This section revisits the examples given in Section 2, and shows how we can define strategies for these examples using the interleaving combinators.

6.1 Applying Tautologies

Figure 1 presents a collection of rules for rewriting logical expressions. We assume that these rules are applied from left to right. Rules for expressing the associativity of conjunction and disjunction are missing: instead, we assume that the given rules are applied modulo the associativity of these operators. The rule set should also be completed by adding commutative variations of the presented rewrite rules (e.g., $F \vee \phi = \phi$ for OrFalse).

The rules in Figure 1 are grouped into categories (such as "negations"), and we use this grouping to combine the rules and categories into strategies:

$$
\begin{array}{lll}
\mathit{negations} & = \textsc{NotNot} & + \textsc{DeMorganAnd} + \textsc{DeMorganOr} \\
\mathit{basics} & = \mathit{constants} & + \mathit{definitions} + \mathit{negations} + \mathit{distribution} \\
\mathit{additionals} & = \mathit{tautologies} + \mathit{contradictions} &
\end{array}
$$

A straightforward approach to reach disjunctive normal form (DNF) is to apply the basic rules exhaustively (the *repeat* combinator), where *somewhere* makes sure that the rules can also be applied to subexpressions:

$$\mathit{dnfExhaustive} = \mathit{repeat}\ (\mathit{somewhere}\ \mathit{basics})$$

Basic Rules:

Constants:
AndTrue: $\phi \wedge T = \phi$ AndFalse: $\phi \wedge F = F$
OrTrue: $\phi \vee T = T$ OrFalse: $\phi \vee F = \phi$
NotTrue: $\neg T = F$ NotFalse: $\neg F = T$

Definitions:
ImplDef: $\phi \rightarrow \psi = \neg\phi \vee \psi$
EquivDef: $\phi \leftrightarrow \psi = (\phi \wedge \psi) \vee (\neg\phi \wedge \neg\psi)$

Negations:
NotNot: $\neg\neg\phi = \phi$
DeMorganAnd: $\neg(\phi \wedge \psi) = \neg\phi \vee \neg\psi$
DeMorganOr: $\neg(\phi \vee \psi) = \neg\phi \wedge \neg\psi$

Distribution:
AndOverOr: $\phi \wedge (\psi \vee \chi) = (\phi \wedge \psi) \vee (\phi \wedge \chi)$

Additional Rules:

Tautologies:
ImplTaut: $\phi \rightarrow \phi = T$ OrTaut: $\phi \vee \neg\phi = T$
EquivTaut: $\phi \leftrightarrow \phi = T$

Contradictions:
AndContr: $\phi \wedge \neg\phi = F$ EquivContr: $\phi \leftrightarrow \neg\phi = F$

Fig. 1. Rules for logical expressions

This strategy is very liberal, and also generates derivations that are less intuitive. The following refined strategy proceeds in four steps, and makes more precise which rule should be applied when, and where.

$$\begin{aligned} \mathit{dnfSteps} = {} & \mathit{label}\ \texttt{"constants"}\ (\mathit{repeat}\ (\mathit{topDown}\ \mathit{constants}\ )) \\ \cdot\ & \mathit{label}\ \texttt{"definitions"}\ (\mathit{repeat}\ (\mathit{somewhere}\ \mathit{definitions}\ )) \\ \cdot\ & \mathit{label}\ \texttt{"negations"}\ (\mathit{repeat}\ (\mathit{topDown}\ \mathit{negations}\ )) \\ \cdot\ & \mathit{label}\ \texttt{"distribution"}\ (\mathit{repeat}\ (\mathit{somewhere}\ \mathit{distribution})) \end{aligned}$$

For example, consider applying strategy *dnfSteps* to $\neg((p \vee q) \rightarrow p)$. This results in the following derivation (and for this logical proposition, no other derivation):

$$\begin{aligned} \neg((p \vee q) \rightarrow p) &= \neg(\neg(p \vee q) \vee p) && \text{(ImplDef)} \\ &= \neg\neg(p \vee q) \wedge \neg p && \text{(DeMorganOr)} \\ &= (p \vee q) \wedge \neg p && \text{(NotNot)} \\ &= (p \wedge \neg p) \vee (q \wedge \neg p) && \text{(AndOverOr)} \end{aligned}$$

If we would have used strategy *dnfExhaustive*, many more derivations would have been allowed, including derivations where $\neg\neg(p \vee q)$ is rewritten into $\neg(\neg p \wedge \neg q)$, giving seven steps in total (instead of just four).

Suppose that we want to extend our strategies for reaching DNF, and also want to use rules for tautologies and contradictions. In the case of *dnfExhaustive*, this can be accomplished by changing its definition into:

$$\mathit{dnfExtra} = \mathit{repeat}\ (\mathit{somewhere}\ (\mathit{basics} + \mathit{additionals}))$$

Changing the original strategy is not always possible, for instance if you want to have the original strategy (without the extra rules) but also the extended strategy (with the extra rules). An alternative approach is to reuse *dnfExhaustive* verbatim, and define *dnfExtra* as follows:

$$dnfExtra = repeat\ (dnfExhaustive + somewhere\ additionals)$$

This definition has some disadvantages too. First of all, the strategy differs considerably from the informal description, and violates the cognitive fidelity principle. Such a difference also influences feedback. Secondly, during the execution of strategy *dnfExhaustive*, this strategy disallows applications of the additional rules. For instance, consider the proposition $p \vee \neg(p \wedge \neg q)$. After one step (taken from *dnfExhaustive*) this is rewritten into $p \vee \neg p \vee \neg\neg q$. At this point, *dnfExhaustive* is not yet finished (because of the double negation), and as a result, the strategy disallows the step with rule OrTaut to $T \vee \neg\neg q$. Extending strategy *dnfExhaustive* using the interleaving combinator is straightforward:

$$dnfExtra = label\ \texttt{"extra"}\ (repeat\ (somewhere\ additionals)) \parallel dnfExhaustive$$

The label in the strategy specification is not necessary, but it provides extra information that can be used for the generation of hints. We return to our earlier example, for which the extended strategy generates two more steps:

$$\begin{array}{lcll} \neg((p \vee q) \rightarrow p) & = & \ldots & \\ & = & (p \wedge \neg p) \vee (q \wedge \neg p) & (\textsc{AndOverOr}) \\ & = & F \vee (q \wedge \neg p) & (\textsc{AndContr}) \\ & = & q \wedge \neg p & (\textsc{OrFalse}) \end{array}$$

Observe the interleaving of steps in this derivation: AndOverOr and OrFalse originate from the *dnfExhaustive* strategy, whereas AndContr comes from the part labeled "extra". Note that this strategy also permits other derivations.

Interestingly, our last definition of *dnfExtra* with interleaving is equivalent to the simpler strategy *repeat* (*somewhere* (*basics* + *additionals*)), and this equivalence can be shown using basic properties of our strategy combinators. To be precise, we need a property explaining how two interleaved repetitions behave[1]: $repeat\ \sigma_1 \parallel repeat\ \sigma_2 = repeat\ (\sigma_1 + \sigma_2)$. Likewise, we use that *somewhere* distributes over choice: $somewhere\ (\sigma_1 + \sigma_2) = somewhere\ \sigma_1 + somewhere\ \sigma_2$. This emphasizes once more the need for having a clear semantics for the combinators.

Strategy *dnfSteps* can also be extended with rules for tautologies and contradictions. Constants are removed in the first step, but the constants introduced by the new rules have to be propagated as well. Hence, the extension to the *dnfSteps* strategy not only adds the new rules, but it also takes care of dealing with the newly introduced constants. The following code fragment shows a possible definition:

[1] Since we have not defined *repeat*, we do not prove this property here. The property also holds for *many* (see Section 3.4).

$$\begin{aligned} \mathit{extension} &= \mathit{repeat}\ (\mathit{somewhere}\ (\mathit{additionals} + \mathit{constants})) \\ \mathit{dnfExtension} &= \mathit{label}\ \texttt{"extension"}\ \mathit{extension} \parallel \mathit{dnfSteps} \end{aligned}$$

Note that this definition is ambiguous in how the constants are removed, because *dnfSteps* can do this (in the first step), but also the extension. This ambiguity has no consequences for the feedback services we offer. The definition gives no priority to the extra rules: they may be used (if possible), but this is not mandatory. Also, it is not required to remove the constants that are introduced by a tautology or contradiction before continuing with bringing the logic expression to DNF.

Our strategy language is expressive enough to specify that constants have to be propagated immediately, including the ones from tautologies and contradictions. For this, we use the atomic combinator in the extension:

$$\mathit{extension} = \mathit{repeat}\ \langle \mathit{somewhere}\ \mathit{additionals} \cdot \mathit{repeat}\ (\mathit{somewhere}\ \mathit{constants}) \rangle$$

6.2 Solving Polynomial Equations of a Higher-Degree

Now that we have interleaving of strategies available, we can adapt the strategy for solving higher-degree equations to make it possible to:

- reuse the strategy for solving quadratic equations;
- follow student behavior even when a student switches from solving one equation to the other;
- give hints about the equation the student is currently solving; and
- show worked-out solutions in which first one of the two quadratic equations is solved, and then the other.

For the latter two points, strategies need to be labeled. The labels are used to determine where in a strategy a student is, and what the corresponding first step would be. Labeling a strategy can either be done automatically, or we can leave this to the strategy developer.

7 Conclusions and Related Work

We have shown how we can add interleaving combinators to our language for specifying rewrite strategies for exercises. We have implemented these combinators in our framework, such that we can follow student behavior, give hints, and show worked-out examples. The implementation of the new combinators, discussed in Section 4, can be translated almost literally to executable Haskell code, the programming language of our choice. The upcoming release of our framework on Hackage[2] will contain the new combinators. Using the interleaving combinators, we can specify strategies closer to textbook description of strategies, allow for more natural student behavior, specify more strategies compositionally, and give various kinds of feedback for strategies using the interleaving combinators. This makes it easier to develop, use, maintain, and reuse strategies.

[2] `http://hackage.haskell.org/package/ideas`

Our strategy language [13] is similar to strategy languages used in computer science and theorem proving [21,5,1] extended with constructs that support giving feedback, such as labels (also present in [1]) and navigation. An interleaving combinator for tactics is easily implemented in a theorem prover such as Isabelle [18]. The interleaving combinators are inspired by the work on communicating sequential processes (CSP) and the algebra of communicating processes (ACP) [14,4], but our goal is to model interactive exercises and to give feedback, instead of modeling concurrent processes. The differences between the ACP approach and our work is the interpretation of a strategy as a parser, which can deal with administrative rules, and the introduction of an operator for specifying atomicity. In contrast with ACP and CSP, we have found no need for adding a communication operator to our language. Our implementation can be seen as an "interleaving parser", which adds an extra level of (interleaving) complexity on top of "permutation parsers" [2], which can be used to parse a number of elements in any order. Current parsing combinator libraries do not offer parser combinators for interleaving parsers[3].

We have implemented interleaving in our framework, but we have yet to gain large-scale experience with the combinators. We will include the interleaving combinator in several of our strategies used within the Math-Bridge project[4], and evaluate the results.

Acknowledgments. This work was made possible by the Math-Bridge project of the Community programme eContent*plus*. The paper does not represent the opinion of the Community, and the Community is not responsible for any use that might be made of information contained in this paper. We thank Alex Gerdes and the anonymous reviewers for commenting on an earlier version of this paper.

References

1. Aspinall, D., Denney, E., Lüth, C.: Tactics for hierarchical proof. Mathematics in Computer Science 3(3), 309–330 (2010)
2. Baars, A.I., Löh, A., Swierstra, S.D.: Parsing permutation phrases. Journal of Functional Programming 14, 635–646 (2004)
3. Beeson, M.J.: Design principles of MathPert: Software to support education in algebra and calculus. In: Kajler, N. (ed.) Computer-Human Interaction in Symbolic Computation, pp. 89–115. Springer, Heidelberg (1998)
4. Bergstra, J.A., Klop, J.W.: Algebra of communicating processes with abstraction. Theoretical Computer Science 37, 77–121 (1985)
5. Bundy, A.: The use of explicit plans to guide inductive proofs. In: International conference on automated deduction, pp. 111–120 (1988)
6. Claessen, K., Hughes, J.: QuickCheck: A lightweight tool for random testing of Haskell programs. In: ICFP 2000, pp. 268–279 (2000)

[3] After discussing our work with Doaitse Swierstra, he implemented a (rather involved) interleaving combinator for parsers on top of his parser combinator library [20].

[4] http://service.math-bridge.org/

7. Cohen, A., Cuypers, H., Reinaldo Barreiro, E., Sterk, H.: Interactive mathematical documents on the web. In: Algebra, Geometry and Software Systems, pp. 289–306. Springer, Heidelberg (2003)
8. Doorman, M., Drijvers, P., Boon, P., van Gisbergen, S., Gravemeijer, K.: Design and implementation of a computer supported learning environment for mathematics. In: Earli 2009 SIG20 invited Symposium Issues in designing and implementing computer supported inquiry learning environments (2009)
9. Fokkink, W.: Introduction to Process Algebra. Springer, Heidelberg (2000)
10. Gerdes, A., Heeren, B., Jeuring, J.: Constructing Strategies for Programming. In: Cordeiro, J., et al. (eds.) Proceedings of the First International Conference on Computer Supported Education, pp. 65–72. INSTICC Press (March 2009)
11. Gerdes, A., Heeren, B., Jeuring, J.: Properties of Exercise Strategies. In: Proceedings of IWS 2010: 1st International Workshop on Strategies in Rewriting, Proving, and Programming. Electronic Proceedings in Theoretical Computer Science (2011)
12. Heeren, B., Jeuring, J.: Adapting mathematical domain reasoners. In: Autexier, S., Calmet, J., Delahaye, D., Ion, P.D.F., Rideau, L., Rioboo, R., Sexton, A.P. (eds.) AISC 2010. LNCS, vol. 6167, pp. 315–330. Springer, Heidelberg (2010)
13. Heeren, B., Jeuring, J., Gerdes, A.: Specifying rewrite strategies for interactive exercises. Mathematics in Computer Science 3(3), 349–370 (2010)
14. Hoare, C.A.R.: Communicating sequential processes. Prentice-Hall, Inc., Englewood Cliffs (1985)
15. Huet, G.: The zipper. Journal of Functional Programming 7(5), 549–554 (1997)
16. Lodder, J., Heeren, B.: A teaching tool for proving equivalences between logical formulae. In: Soler-Toscano, F. (ed.) TICTTL 2011. LNCS, vol. 6680, pp. 154–161. Springer, Heidelberg (2011)
17. Melis, E., Siekmann, J.: ActiveMath: An intelligent tutoring system for mathematics. In: Rutkowski, L., Siekmann, J.H., Tadeusiewicz, R., Zadeh, L.A. (eds.) ICAISC 2004. LNCS (LNAI), vol. 3070, pp. 91–101. Springer, Heidelberg (2004)
18. Nipkow, T., Paulson, L.C., Wenzel, M.T. (eds.): Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
19. Peyton Jones, S., et al.: Haskell 98, Language and Libraries. The Revised Report. Cambridge University Press, A special issue of the Journal of Functional Programming, (2003), http://www.haskell.org/
20. Swierstra, S.D.: Combinator parsing: A short tutorial. In: Bove, A., Barbosa, L.S., Pardo, A., Pinto, J.S. (eds.) Language Engineering and Rigorous Software Development. LNCS, vol. 5520, pp. 252–300. Springer, Heidelberg (2009)
21. Visser, E., Benaissa, Z.A., Tolmach, A.: Building program optimizers with rewriting strategies. In: ICFP 1998, pp. 13–26 (1998)

Combining Source, Content, Presentation, Narration, and Relational Representation

Fulya Horozal, Alin Iacob, Constantin Jucovschi,
Michael Kohlhase, and Florian Rabe

Computer Science, Jacobs University, Bremen
{f.horozal,a.iacob,c.jucovschi,m.kohlhase,f.rabe}@jacobs-university.de

Abstract. In this paper, we try to bridge the gap between different dimensions/incarnations of mathematical knowledge: MKM representation formats (content), their human-oriented languages (source, presentation), their narrative linearizations (narration), and relational presentations used in the semantic web. The central idea is to transport solutions from software engineering to MKM regarding the parallel interlinked maintenance of the different incarnations. We show how the integration of these incarnations can be utilized to enrich the authoring and viewing processes, and we evaluate our infrastructure on the LATIN Logic Atlas, a modular library of logic formalizations, and a set of computer science lecture notes written in SᴛᴇX – a modular, semantic variant of LaTeX.

1 Introduction

Within the Mathematical Knowledge Management (MKM) community, XML-based content representations of mathematical formulae and knowledge have been developed that are optimized for machine processing. They serve as archiving formats, make mathematical software systems and services interoperable and allow to develop structural services like search, documentation, and navigation that are independent of mathematical foundations and logics.

However, these formats are by their nature inappropriate for human processing. Therefore, community uses languages that are less verbose, more mnemonic, and often optimized for a specific domain for authoring. Such human-oriented languages (we call them source languages) are converted — via a complex compilation process — into the content representations for interaction with MKM services, ideally without the user ever seeing them. In addition, we have designed presentation-oriented languages that permit an enriched reading experience compared to the source language.

This situation is similar to software engineering, where programmers write code, run the compiled executables, build HTML-based API documentations, but expect, *e.g.*, the documentation and the results of debugging services in terms of the sources. In software engineering, scalable solutions for this problem have been developed and applied successfully, which we want to transfer to MKM.

J.H. Davenport et al. (Eds.): Calculemus/MKM 2011, LNAI 6824, pp. 212–227, 2011.

The work described here originates from our work on two large collections of mathematical documents: our LATIN Logic Atlas [KMR09] formalized in the logical framework LF; and our General Computer Science lecture notes written in STEX [Koh08] – a modular, semantic variant of LATEX. Despite their different flavor, both collections agree in some key aspects: They are large, highly structured, and extensively inter-connected, and both authoring and reading call for machine support.

Moreover, they must be frequently converted between representation dimensions optimized for different purposes: a human-friendly input representation (source), a machine-understandable content markup (content), interactive documents for an added-value reading experience (presentation-content parallel markup), a linearized structure for teaching and publication (narration), and a network of linked data items for indexing and integration with the semantic web (relational).

Therefore, we see a need for a representation and distribution format that specifies these dimension and enables their seamless integration. In this paper, we present one such format and evaluate it within our system and document collections. In Sect. 2, we first give an overview over the document collections focusing on the challenges they present to knowledge management. Then we design our representation format in Sect. 3. In Sect. 4 and 5, we show how we leverage this methodology in the authoring and the viewing process.

2 Structured Document Collections

2.1 The LATIN Logic Atlas

The LATIN Logic Atlas [KMR09] is a library of formalizations of logics and related formal systems as well as translations between them. It is intended as a reference and documentation platform for logics commonly used in mathematics and computer science. It uses a foundationally unconstrained logical framework based on modular LF and its Twelf implementation [HHP93, PS99, RS09] and focuses on modularity and extensibility.

The knowledge in the Logic Atlas is organized as a graph of LF signatures and signature morphisms between them. The latter are split into inheritance translations (inclusions/imports) and representation theorems, which have to be proved. It contains formalizations of type theories, set theories, logics and mathematics. Among the logic formalizations in the Logic Atlas are, for example, the formalizations of propositional (*PL*), first (*FOL*) and higher-order logic (*HOL*), sorted (*SFOL*) and dependent first-order logic (*DFOL*), description logics (*DL*), modal (*ML*) and common logic (*CL*) as illustrated in Fig. 1. Single arrows ($\rightarrow$) in this diagram denote translations between formalizations and hooked arrows ($\hookrightarrow$) denote imports.

All logics are designed modularly formed from orthogonal theories for individual connectives, quantifiers, and axioms. For example, the classical $\wedge$ connective is only declared once in the whole Logic Atlas, and the axiom of excluded middle and its consequences reside in a separate signature.

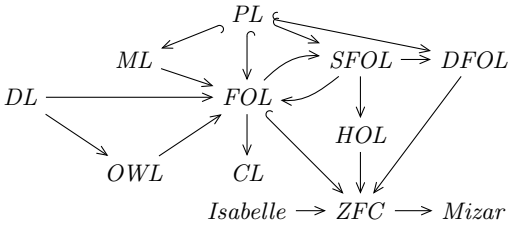


Fig. 1. Logical Languages in the LATIN Logic Atlas

As a running example, we introduce a very simple fragment of the formalization of the syntax of propositional logic.

Example 1 (Propositional Logic) The formalization of propositional logic syntax consists of the LF signatures illustrated in Fig. 2. We focus on the structural aspects and omit the details of LF. We define four signatures living in two different namespaces. `BASE` declares a symbol for the type of propositions. It is imported into `CONJ` and `IMP` which declare conjunction and implication, respectively, and these are imported to `PROP`.

```
%namespace = "http://cds.omdoc.org/logics"
%sig BASE = {
 o : type. %% Type of propositions
}

%namespace = "http://cds.omdoc.org/logics/propositional"
%sig CONJ = {%include BASE. ...}
%sig IMP  = {%include BASE. ...}
%sig PROP = {%include CONJ. %include  IMP.}
```

Fig. 2. Formalization of Propositional Logic Syntax in Twelf

Overall, the Logic Atlas contains over 750 LF signatures, and their highly modular structure yields a large number of inheritance edges. Additionally, it contains over 500 signature morphisms that connect the nodes. This leads to a highly interlinked non-linear structure of the Logic Atlas. Moreover, it is designed highly collaboratively with strong interdependence between the developers. Therefore, it leads to a number of MKM challenges.

Firstly, the LF modules are distributed over files and these files over directories. This structure is semantically transparent because all references to modules are made by URIs. The URIs are themselves hierarchical grouping the modules into nested namespaces. It is desirable that these namespaces do not have to correspond to source files or directories. Therefore, a mapping between URIs and URLs has to be maintained and be accessible to all systems.

Secondly, LF encodings are usually highly concise, and a complex type reconstruction process is needed to infer additional information. Twelf can generate

an OMDoc-based content representation of the sources, but this semantically enriched version is currently not fed back into the editing process.

Thirdly, encodings in LF are often difficult to read for anybody but the author. In a collaborative setting, it is desirable to interact with the Logic Atlas not only through the LF source syntax but also through browsable, cross-referenced XHMTL+MathML. These should be interactive and for example permit looking up the definition of a symbol or displaying the reconstructed type of a variable. While we have presented such an interface in [GLR09] already, the systematic integration into the authoring process, where the state of the art is a text editor, has so far been lacking.

Finally, to understand and navigate the Logic Atlas, it is necessary to visualize its multi-graph structure in an interactive graphical interface.

2.2 Computer Science Lecture Notes in Planetary

The GenCS corpus consists of the course notes and problems of a two-semester introductory course in Computer Science [Gen11] held at Jacobs University by one of the authors in the last eight years. The course notes currently comprise 300 pages with over 500 slides organized in over 800 files; they are accompanied by a database of more than 1000 homework/exam problems. All course materials are authored and maintained in the sTEX format [Koh08], a modular, semantic variant of LaTeX that shares the information model with OMDoc; see Fig. 3 for an example.

```
\begin{module}[id=trees]
  \symdef[name=tdepth]{tdepthFN}{\text{dp}}
  \symdef{tdepth}[1]{\prefix\tdepthFN{#1}}
  \begin{definition}[id=tree-depth.def]
    Let $\defeq{T}{\tup{V,E}}$ be tree, then the {\definiendum [tree-depth]{depth}}
    $\tdepth{v}$ of a node $\inset{v}{V}$ is defined recursively: $\tdepth{r}=0$ for
    the root $r$ of $T$ and $\tdepth(w)=1+\tdepth(w)$ if $\inset{\tup{v,w}}E$.
 \end{definition}
 ...
\end{module}

\begin{module}[id=binary-trees]
  \importmodule[\KWARCslides{graphs-trees/en/trees}]{trees}
  ...
  \begin{definition}[id=binary-tree.def,title=Binary Tree]
    A \definiendum[binary-tree]{binary tree} is a \termref[cd=trees,name=tree]{tree}
    where all \termref[cd=graphs-intro,name=node]{nodes}
    have \termref[cd=graphs-intro,name=out-degree]{out-degree} 2 or 0.
  \end{definition}
  ...
\end{module}
```

Fig. 3. Semiformalization of two course modules

In our nomenclature, sTEX is used as a source language that is transformed into OMDoc via the LaTeXML daemon [GSK11]. For debugging and high-quality print the sTEX sources can also be typeset via `pdflatex`, just as ordinary LaTeX documents. The encoding makes central use of the modularity afforded by the theory graph approach; knowledge units like slides are encoded as "modules"

(theories in OMDoc) and are interconnected by theory morphisms (module imports). Modules also introduce concepts via \definiendum and semantic macros via \symdef, these are inherited via the module import relation.

The challenges discussed for the LATIN Logic Atlas apply to the GenCS corpus and the Planetary system analogously. Moreover: (i) The development of the corpus proceeds along two workflows: drafting of the content via the classical pdflatex conversion (in a working copy via make) and via the web interface in the Planetary system. (ii) The LaTeXML conversion process needs intermediate files (the sTeX module signatures), all of which have to be kept in sync. (iii) Importing legacy sTeX materials into the Planetary system is nontrivial. For example, the contents of http://planetmath.org contain many (semantic) cross-links and metadata. Rather than a set of individualized import scripts at the level of the repositories and databases underlying Planetary, it is desirable to have a file (set) that can be imported uniformly.

3 A Multi-dimensional Knowledge Representation

3.1 Dimensions of Knowledge

In order to address the knowledge management challenges outlined above, we devise a methodology that permits the parallel maintenance of the orthogonal dimensions of the knowledge contained in a collection of mathematical documents. It is based on two key concepts: (i) a hierarchic organization of dimensions and knowledge items in a file-system-like manner, and (ii) the use of MMT URIs [RK11] as a standardized way to interlink both between different knowledge items and between the different dimensions of the same knowledge item.

The MMT URI of a toplevel knowledge item is of the form $g?M$ where g is the namespace and M the module name. Namespaces are URIs of the form ⟪scheme⟫ `://[`⟪userinfo⟫`@]`$D_1 \ldots D_m$`[:`⟪port⟫`]/`$S_1/\ldots/S_n$ where the D_i are domain labels and the S_i are path segments. Consequently, $g?M$ is a well-formed URI as well. ⟪userinfo⟫, and ⟪port⟫ are optional, and ⟪userinfo⟫, ⟪scheme⟫, and ⟪port⟫ are only permitted so that users can form URIs that double as URLs — MMT URIs differing only in the scheme, userinfo, or port are considered equal.

We arrange a collection of mathematical documents as a folder containing the following subfolders, all of which are optional:

source contains the source files of a project. This folder does not have a predefined structure.

content contains a semantically marked up representation of the source files in the OMDoc format. Every namespace is stored in one file whose path is determined by its URI. Modules with namespace $D_1. \ldots .D_m/S_1/.../S_n$ reside in an OMDoc file with path content/$D_m/\ldots/D_1/S_1/\ldots/S_n$`.omdoc`. Each module carries an attribute `source="/PATH?colB:lineB-colE:lineE"` giving its physical location as a URL. Here PATH is the path to the containing file in the source, and colB, lineB, colE, and lineE give the begin/end column/line information.

`presentation` contains the presentation of the source files in the XHTML+MathML format with JOBAD annotations [GLR09]. It has the same file structure as the folder `content`. The files contain XHTML elements whose body has one child for every contained module. Each of these module has the attribute `jobad:href="URI"` giving its MMT URI.

`narration` contains an arbitrary collection of narratively structured documents. These are OMDoc files that contain narrative content such as sectioning and transitions, but no modules. Instead they contain reference elements of the form `<mref target="MMTURI"/>` that refer to MMT modules. It is common but not necessary that these modules are present in the `content` folder.

`relational` contains two files containing an RDF-style relational representation of the content according to the MMT ontology. Both are in XML format with toplevel element `mmtabox` and a number of children. In `individuals.abox`, the children give instances of unary predicates such as `<individual type="IsTheory" uri="MMTURI" source="PATH"/>`. In `relations.abox`, the children give instances of binary predicates such as `<relation subject="MMTURI1" predicate="ImportsFrom" object="MMTURI2" source="PATH"/>`. Usually, the knowledge items occurring in unary predicates or as the subject of a binary predicate are present in the content. However, the object of a binary predicate is often not present, namely when a theory imports a remote theory. In both cases, we use an attribute `source` to indicate the source that induced the entry; this is important for change management when one of the source files was changed.

Example 2 (Continuing Ex. 1)
The directory structure for the signatures from Ex. 1 is given in Fig. 4 using a root folder named `propositional-syntax`. Here we assume that the subfolder `source` contains the Twelf source files `base.elf` which contains the signature `BASE`, `modules.elf` which contains `CONJ` and `IMP`, and `prop.elf` which contains `PROP`.

Based on the MMT URIs of the signatures in the source files, their content representation is given as follows. The signature `BASE` has the MMT URI `http://cds.omdoc.org/logics?BASE`. The other signatures have MMT URIs such as `http://cds.omdoc.org/logics/propositional/syntax?CONJ`. The content representation of the signature `BASE` is given in the OMDoc file `content/org/omdoc/cds/logics.omdoc`. The other content representations reside in the file `content/org/omdoc/cds/logics/propositional/syntax.omdoc`.

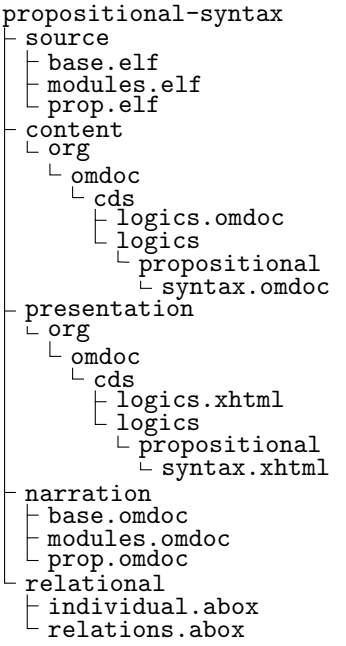


Fig. 4. Folder Structure

The subfolder `presentation` contains the respective XHTML files, `logics.xhtml` and `syntax.xhmtl`. All files in Fig. 4 can be downloaded at `https://svn.kwarc.info/repos/twelf/projects/propositional-syntax`.

Our methodology integrates various powerful conceptual distinctions that have been developed in the past. Firstly, our distinction between the source and the content representation corresponds to the distinction between source and binary in software engineering. Moreover, our directory structure is inspired by software projects, such as in Java programming. In particular, the use of URIs to identify content (binary) items corresponds to identifiers for Java classes. Therefore, existing workflows and implementations from software engineering can be easily adapted, for example in the use of project-based IDEs (see Sect. 4).

Secondly, the distinction between content and presentation has been well studied in the MKM community and standardized in MathML [ABC+03]. In particular, the cross-references from presentation to content correspond to the interlinking of content and presentation in the parallel markup employed in MathML, which we here extend to the level of document collections.

Thirdly, the distinction between content and narrative structure was already recognized in the OMDoc format. The general intuition there is that narrative structures are "presentations" at the discourse level. But in contrast to the formula level, presentations cannot be specified and managed via notation definitions. Instead we add narrative document structure fragments, i.e. document-structured objects that contain references to the content representations and transition texts as lightweight structures to the content commons; see [Mül10] for details and further references.

Finally, the distinction between tree-structured content representation and the relational representation corresponds to the practice of the semantic web where RDF triples are used to represent knowledge as a network of linked data.

3.2 A Mathematical Archive Format

We will now follow the parallelism to software engineering developed in the previous section: We introduce mathematical archives — `mar` files — that correspond to Java archives, i.e., `jar` files [Ora]. We define a mathematical archive to be a zip file that contains the directory structure developed in Sect. 3.1. By packaging all knowledge dimensions in a single archive, we obtain a convenient and lightweight way of distributing multi-dimensional collections of interlinked documents.

To address into the content of a `mar` archive, we also define the following URL scheme: Given a `mar` whose URL is `file:/A` and which contains the source file `source/S`, then the URL `mar:/A/S` resolves to that source file. We define the URL `mar:/A/S?Pos` accordingly if `Pos` is the position of a module given by its line/column as above.

Similar to the compilation and building process that is used to create `jar` files, we have implemented a building process for `mar` files. It consists of three stages. The first stage (compilation) depends on the source language and produce one

OMDoc file for every source file whose internal structure corresponds to the source file. This is implemented in close connection with dedicated tools for the source language. In particular, we have implemented a translation from LF to OMDoc as part of the Twelf implementation of LF [PS99, RS09]. Moreover, we have implemented a translation from STEX to OMDoc based on the LATEXML daemon [GSK11].

The second stage (building) is generic and produces the remaining knowledge dimensions from the OMDoc representation. In particular, it decomposes the OMDoc documents into modules and reassembles them according to their namespaces to obtain the content representation. The narrative dimension is obtained from the initial OMDoc representation by replacing all modules with references to the respective content item. We have implemented this as a part of the existing MMT API [RK11]. Finally, the API already includes a rendering engine that we use to produce the presentation and the relational representation.

Then the third stage (packaging) collects all folders in a zip archive. For LF, we integrate all three stages into a flexible command line application.

Example 3 (Continuing Ex. 2) The mathematical archive file for the running example can be obtained at `https://svn.kwarc.info/repos/twelf/projects/propositional-syntax.mar`

3.3 Catalog Services

The use of URIs as knowledge identifiers (rather than URLs) is crucial in order to permit collaborative authoring and convenient distribution of knowledge. However, it requires a catalog that translates an MMT URIs to the physical location, given by a URL, of a resource. Typical URLs are those in a file system, in a mathematical archive, or a remote or local repository. It is trivial to build the catalog if the knowledge is already present in content form where locations are derived from the URI.

But the catalog is already needed during the compilation process: For example, if a theory imports another theory, it refers to it by its MMT URI. Consequently, the compilation tool must already be aware of the URI-to-URL mapping before the content has been produced. However, the compilation tool is typically a dedicated legacy system that natively operates on URLs already and does not even recognize URIs. This is the case for both Twelf and LATEX.

Therefore, we have implemented standalone catalog services for these two tools and integrated them with the respective system. In the case of Twelf, the catalog maintains a list of local directories, files, and `mar` archives that it watches. It parses them whenever they change and creates the URI-URL mapping. When Twelf encounters a URI, it asks the catalog via HTTP for the URL. This parser only parses the outer syntax that is necessary to obtain the structure of the source file; it is implemented generically so that it can be easily adapted to other formal declarative languages. In this sense, it is similar to the HTTP Getter service of the HELM library [ASPC+03].

An additional strength of this catalog is that it can also handle ill-formed source representations that commonly arise during authoring. Moreover, we also use the catalog to obtain the line/column locations of the modules inside the source files so that the content-to-source references can be added to the content files.

In the case of sTEX, a poor man's catalog services is implemented directly in TEX: the base URIs of the GenCS knowledge collection (see `\KWARCslides` in Fig 3) is specified by a `\defpath` statement in the document preamble and can be used in the `\importmodule` macros. The `module` environments induce internal TEX structures that store information about the imports (`\importmodule`) structure and semantic macros (`\symdef`), therefore these three sTEX primitives have to be read whenever a module is imported. To get around difficulties with selective input in TEX, the sTEX build process excerpts a sTEX signature module ⟪`module`⟫`.sms` from any module ⟪`module`⟫`.tex`. So `\importmodule`⟪`module`⟫`.sms` simply reads ⟪`module`⟫`.sms`.

4 The Author's Perspective

The translation from source to a content-like representation has been well-understood. For languages like LF, it takes the form of a parsing and type reconstruction process that transforms external to internal syntax. The translation from internal syntax to an OMDoc-based content representation is conceptually straightforward. However, it is a hard problem to use the content representation to give the author feedback about the document she is currently editing. A more powerful solution is possible if we always produce all knowledge dimensions using the compilation and building process as described in Sect. 3.2. Then generic services can be implemented easily, each of them based on the most suitable dimension, and we give a few examples in Sect. 4.2.

Note that this is not an efficiency problem: Typically the author only works on a few files that can be compiled constantly. It is even realistic to hold all dimensions in memory. The main problem is an architectural one, which is solved by our multi-dimensional representation. Once this architecture is set up and made available to IDE developers, it is very easy for them to quickly produce powerful generic services.

4.1 Multi-dimensional Knowledge in an IDE

In previous work, we have already presented an example of a semantic IDE [JK10] based on Eclipse. We can now strengthen it significantly by basing it on our multi-dimensional representation. Inspired by the project metaphor from software engineering, we introduce the notion of a mathematical project in Eclipse.

A mathematical project consists of a folder containing the subfolder from Sect. 3.1. The author works on a set of source files in the `source` directory. Moreover, the project maintains a mathpath (named in analogy to Java's classpath) that provides a set of `mar` archives that the user wishes to include.

The IDE offers the build functionality that runs the compilation and building processes described in Sect. 3.2 to generate the other dimensions from the source dimension. The key requirement here is to gracefully degrade in the presence of errors in the source file. Therefore, we provide an adaptive parser component that consists of three levels:

The **regex level** uses regular expressions to spot important structural properties in the document (e.g. the namespace and signature declarations in the case of LF). This compilation level never fails, and its result is an OMDoc file that contains only the spotted structures and lacks any additional information of the content.

The **CFG parser level** uses a simple context-free grammar to parse the source. It is able to spot more complicated structures such as comments and nested modules and can be implemented very easily within Eclipse. Like the previous level, it produces an approximate OMDoc file, but contrary to the previous level, it may find syntax errors that are then displayed to the user.

The **full parser level** uses the dedicated tool (Twelf or LaTeX). The resulting OMDoc file includes the full content representation. In particular, in the case of Twelf, it contains all reconstructed types and implicit arguments. However, it may fail in the case of ill-typed input.

The adaptive parser component tries all parser in stages and retains the file returned by the last stage that succeeds. This file is then used as the input to produce the remaining dimensions.

4.2 Added-Value Services

In this section we present several services that aim at supporting the authoring process. We analyze each of these services and show that they can can be efficiently implemented by using one or several dimensions of knowledge.

project explorer is a widget giving an integrated view on a project's content by abstracting from the file system location where the sources are are defined. It groups objects by their content location, i.e., their MMT URI. To implement this widget, we populate the non-leaf nodes of the tree from the directory structure of the content dimension. The leaf nodes are generated by running simple XPath queries on the OMDoc files.

outline view is a source level widget which visualizes the main structural components. For LF, these include definitions of signatures and namespaces as well as constant declarations within signatures. Double-clicking on any such structural components opens the place in the source code where the component is defined. Alternatively, the corresponding presentation can be opened.

autocompletion assists the user with getting location and context specific suggestions, e.g., listing declarations available in a namespace. Fig. 5a) shows an example. Note how the namespace prefix `base` is declared to point to a certain namespace, and the autocompletion suggests only signatures names declared in that namespace. The implementation of this feature requires information about the context where autocompletion is requested, which is

obtained from the interlinked source and content dimensions. Moreover, it needs the content dimension to compute all possible completions. In more complicated scenarios, it can also use the relational dimension to compute the possible completions using the relational queries.

hover overlay is a feature that shows in-place meta-data about elements at the position of the mouse cursor such as the full URIs of a symbol, its type or definitions, a comment, or inferred types of variables. Fig. 5b) shows an example. The displayed information is retrieved from the content dimension. It is also possible to display the information using the presentation dimension.

definition/reference search makes it easy for a user to find where a certain item is defined or used. Although the features require different user interfaces the functionality is very similar, namely, finding relations. Just like in the hover overlay feature, one first finds the right item in the content representation and then use the relation dimension to find the requested item(s).

theory-graph display provides a graphical visualization of the relations among knowledge items. To implement this feature we apply a filter on the multigraph from the relations dimension and use 3rd party software to render it.

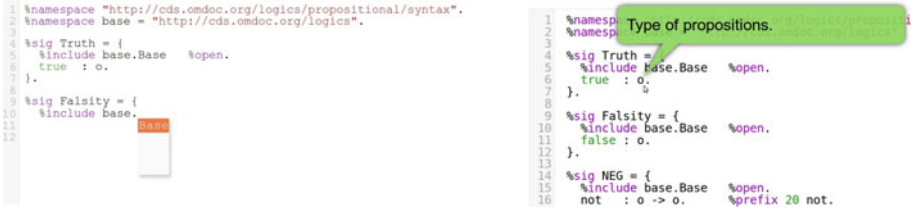


Fig. 5. a) Context aware Auto-Completion b) Metadata information on hover

5 The Reader's Perspective

We have developed the Planetary system [KDG+11] as the reader's complement to our IDE. Planetary is a Web 3.0 system[1] for semantically annotated document collections in Science, Technology, Engineering and Mathematics (STEM). In our approach, *documents published in the Planetary system become flexible, adaptive interfaces to a content commons* of domain objects, context, and their relations.

We call this framework the **Active Documents Paradigm** (ADP), since documents can also actively adapt to user preferences and environment rather than only executing services upon user request. Our framework is based on *semantically annotated documents* together with semantic background ontologies (which we call the **content commons**). This information can then be used by user-visible, semantic services like program (fragment) execution, computation, visualization, navigation, information aggregation and information retrieval [GLR09].

[1] We adopt the nomenclature where Web 3.0 stands for extension of the Social Web with Semantic Web/Linked Open Data technologies.

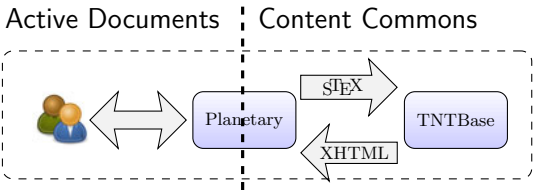
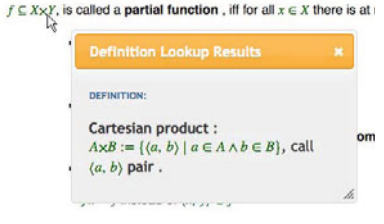


Fig. 6. The Active Documents Architecture

The Planetary system directly uses all five incarnations/dimensions of mathematical knowledge specified in Sect 3.1. In the `content` incarnation, Planetary uses OMDoc for representing content modules, but authors create and maintain these using sTEX in the `source` dimension and the readers interact with the active documents encoded as dynamic XHTML+MathML+RDFa (the `source` incarnation of the material). We use the LaTeXML daemon [Mil, GSK11] for the transformation from sTEX to OMDoc, this is run on every change to the sTEX sources. The basic presentation process [KMR08] for the OMDoc content modules is provided by the TNTBase system [ZK09b].

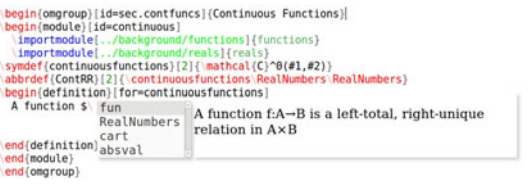
But Planetary also uses the `narrative` dimension: content modules are used not only for representing mathematical theories, but also for document structures: **narrative modules** consist of a mixture of sectional markup, inclusion references, and narrative texts that provide transitions (narrative glue) between the other objects. Graphs of narrative modules whose edges are the inclusion references constitute the content representations of document fragments, documents, and document collections in the Planetary system, wich generates active documents from them. It uses a process of separate compilation and dynamic linking to equip them with document (collection)-level features like content tables, indexes, section numbering and inter-module cross-references; see [DGK+11] for details.

As the active documents use identifiers that are relative to the base URI of the Planetary instance, whereas the content commons uses MMT URIs, the **semantic publishing map** which maintains the correspondence between these is a central, persistent data structure maintained by the Planetary system.



This is one instance of the `relational` dimension, another is used in the For instance, the RDFa embedded in the presentation of a formula (and represented in the linking part of the math archive) can be used for definition lookup as shown on the left. Actually the realization of the definition lookup service involves `presentation` (where the service is embedded) and `content` (from which the definition is fetched to be presented by the service) incarnations as well.

In the future we even want to combine this with the `source` dimension by combining it with a `\symdef`-look service that makes editing easier. This can be thought of as a presentation-triggered complement to the editor-based service on the right that looks up `\symdef`s by their definienda.



6 Discussion

Our approach of folder-based projects is inspired by practices from software engineering. However, while it is appealing to port these successful techniques to mathematics, mathematical knowledge has more dimensions with a more complex inter-dependence than software. In particular, where software IDEs can be optimized for a work flow of generating either binaries or documentation from the sources, the build process for our mathematical projects must employ multiple stages. Moreover, the narrative dimension that is central to mathematics has no direct analog in software engineering.

Therefore, we cannot expect to be able to adapt all software engineering practices directly. We have been very successful with authoring-based services, where the IDE can utilize the various dimensions for added-value services. Contrary to software projects, this often requires the integration of information from multiple dimensions. Archive-based packaging and distribution work flows can also be adapted successfully, and we expect that the same holds for dependency management and — along the lines of [DGK+11] — linking.

More difficulties are posed by authoring work flows that depend on the mathematical semantics of the content such as generating content (e.g., theorem proving) or highlighting errors while typing. The necessary integration of the user interface with a deduction system is very difficult, and only few systems realize it. For example, the Agda [Nor05] emacs mode can follow cross-references and show reconstructed types of missing terms. The Isabelle jEdit user interface [Wen10] can follow cross-references and show tooltips derived from the static analysis. In general, we believe that a data model like ours is one ingredient in the design of a comprehensive and scalable solution for this problem.

Various domain-specific systems have adopted ad hoc implementations of multi-dimensional representations. For example, the Mizar workflow [TB85] can be understood as a build process that transforms the source dimension into various content and relational dimensions, which are then used to assemble an enriched content (.abs) and a presentation dimension (.html). Similar to our format, the dimensions are spread over several folders. The Archive of Formal Proofs [KNe04] uses a source and two narrative dimensions (proof outline and document in pdf format) along with an HTML-based presentation dimension. The dimensions are not systematically separated into folders, and projects can be distributed as tar.gz archives.

"Mathematica Applications" distinguish dimensions for mathematical algorithms (kernel), notations (notebooks with palettes front-end), and narration (narrative notebooks in the documentation). They also allow pre-generating XHTML representations of the documentation (without giving it a separate dimension). The documentation notebooks can contain (non-semantic) links. The algorithm aspect — that is at the center of Mathematica — is missing from our math archives as content representations of mathematical computation are largely unstandardized in MKM. Conversely, the separation of the source, content, and presentation dimensions are missing from Mathematica Applications because they are conflated.

We have so far evaluated our infrastructure using our own corpora and systems: two corpora in two source languages with the respective compilation processes (LATIN in Twelf and lecture notes in STEX), our semantic IDE [JK10], and our semantic publishing system [KDG+11]. Despite the close collaboration of the respective developers and users, we have found that a representation format like ours is a key prerequisite for scalable system integration: We use a mathematical archive as a central knowledge store around which all systems are arranged — with each system producing and/or using some of the knowledge dimensions. For example, the navigation of the LATIN Logic Atlas involves four distinct processes that can only be connected through our format: converting *source* to *content*, generating *relational* from *content*, using *relational* to compute and display the graph, using the cross-references to switch focus in the *source* editor based on the user interaction with the graph.

Our choice of dimensions is not final — we have focused on those that we found most important. Other dimensions can be added to our format easily, for example, discussions and reviews or additional presentation dimensions like pdf. Planetary already allows discussions of individual content items, and STEX already permits the creation of pdf. A more drastic extension is the use of multiple source dimensions, e.g., for a specification in STEX, a formalization in a proof assistant, and algorithms implemented in a programming language. Here the MMT URIs will be particularly useful to realize cross-references across dimensions.

Our emphasis on file systems and zip archives is not without alternative. In fact, even our own LATIN Logic Atlas is stored in a custom XML database with SVN interface [ZK09a]; Planetary uses a custom database as well. However, a file system-based infrastructure is the easiest way to specify a multi-dimensional representation format and represents the lowest hurdle for system integration. Moreover, we believe that more sophisticated mathematical databases should always be able to import and export knowledge in a format like ours.

7 Conclusion

We have presented an infrastructure for creating, storing, managing, and distributing mathematical knowledge in various pragmatic forms. We have identified five aspects that have to be taken into account here: (i) human-oriented *source languages* for efficient editing of content, (ii) the modular *content representation* that has been the focus of attention in MKM so far, (iii) *interactive*

presentations of the content for viewing, navigation, and interaction, (iv) *narrative structures* that allow binding the content modules into self-contained documents that can be read linearly, and (v) *relational structures* that cross-link all these aspects and permit keeping them in sync.

These aspects are typically handled by very different systems, which makes system integration difficult, often leading to ad-hoc integration solutions. By designing a flexible knowledge representation format featuring multiple interconnected dimensions, we obtained a scalable, well-specified basis for system integration. We have evaluated them from the authoring and reading perspectives using two large structured corpora of mathematical knowledge.

In the future, we want to combine these systems and perspectives more tightly. For example, we could use Planetary to discuss and review logic formalizations in Twelf, or write papers about the formalizations in STEX. This should not pose any fundamental problems as the surface languages are interoperable by virtue of having the same, very general data model: the OMDoc ontology. By the same token we want to add additional surface languages and presentation targets that allow to include other user groups. High-profile examples include the Mizar Mathematical Language and Isabelle/Isar.

References

[ABC+03] Ausbrooks, R., Buswell, S., Carlisle, D., Dalmas, S., Devitt, S., Diaz, A., Froumentin, M., Hunter, R., Ion, P., Kohlhase, M., Miner, R., Poppelier, N., Smith, B., Soiffer, N., Sutor, R., Watt, S.: Mathematical Markup Language (MathML) Version 2.0 (2nd edn.) Technical report, World Wide Web Consortium (2003), http://www.w3.org/TR/MathML2

[ASPC+03] Asperti, A., Padovani, L., Coen, C.S., Guidi, F., Schena, I.: Mathematical knowledge management in HELM. Annals of Mathematics and Artificial Intelligence, Special Issue on Mathematical Knowledge Management 38(1-3), 27–46 (2003)

[DGK+11] David, C., Ginev, D., Kohlhase, M., Matican, B., Mirea, S.: A framework for modular semantic publishing with separate compilation and dynamic linking. In: Castro, A.G., Lange, C., Sandhaus, E., de Waard, A. (eds.) 1st Workshop on Semantic Publication, SePublica (2011) (accepted)

[Gen11] General Computer Science: GenCS I/II Lecture Notes (2011), http://gencs.kwarc.info/book/1

[GLR09] Gičeva, J., Lange, C., Rabe, F.: Integrating Web Services into Active Mathematical Documents. In: Carette, J., Dixon, L., Sacerdoti Coen, C., Watt, S. (eds.) MKM 2009, Held as Part of CICM 2009. LNCS, vol. 5625, pp. 279–293. Springer, Heidelberg (2009)

[GSK11] Ginev, D., Stamerjohanns, H., Kohlhase, M.: TheLATEXML daemon: Editable math on the collaborative web. Intelligent Computer Mathematics (2011) (accepted)

[HHP93] Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. Journal of the Association for Computing Machinery 40(1), 143–184 (1993)

[JK10] Jucovschi, C., Kohlhase, M.: sTeXIDE: An Integrated Development Environment for sTeX Collections. In: Autexier, S., Calmet, J., Delahaye, D., Ion, P., Rideau, L., Rioboo, R., Sexton, A. (eds.) AISC 2010. LNCS, vol. 6167, pp. 336–344. Springer, Heidelberg (2010)

[KDG+11] Kohlhase, M., David, C., Ginev, D., Kohlhase, A., Lange, C., Matican, B., Mirea, S., Zholudev, V.: The Planetary System: Web 3.0 & Active Documents for STEM. To appear at ICCS 2011 (finalist at the Executable Papers Challenge) (2011), https://svn.mathweb.org/repos/planetary/doc/epc11/paper.pdf

[KMR08] Kohlhase, M., Müller, C., Rabe, F.: Notations for living mathematical documents. In: Autexier, S., Campbell, J., Rubio, J., Sorge, V., Suzuki, M., Wiedijk, F. (eds.) AISC 2008, Calculemus 2008, and MKM 2008. LNCS (LNAI), vol. 5144, pp. 504–519. Springer, Heidelberg (2008)

[KMR09] Kohlhase, M., Mossakowski, T., Rabe, F.: The LATIN Project (2009), https://trac.omdoc.org/LATIN/

[KNe04] Klein, G., Nipkow, T., Paulson, L. (eds.) Archive of Formal Proofs (2004), http://afp.sourceforge.net/

[Koh08] Kohlhase, M.: Using LaTeX as a semantic markup format. Mathematics in Computer Science 2(2), 279–304 (2008)

[Mil] Miller, B.: LaTeXML: A LaTeX to XML converter

[Mül10] Müller, C.: Adaptation of Mathematical Documents. PhD thesis, Jacobs University Bremen (2010)

[Nor05] Norell, U.: The Agda WiKi (2005), http://wiki.portal.chalmers.se/agda

[Ora] Oracle. JDK 6 Java Archive (JAR), http://download.oracle.com/javase/6/docs/technotes/guides/jar

[PS99] Pfenning, F., Schürmann, C.: System description: Twelf - A meta-logical framework for deductive systems. In: Ganzinger, H. (ed.) CADE 1999. LNCS (LNAI), vol. 1632, pp. 202–206. Springer, Heidelberg (1999)

[RK11] Rabe, F., Kohlhase, M.: A Scalable Module System. Under review (2011), http://arxiv.org/abs/1105.0548

[RS09] Rabe, F., Schürmann, C.: A Practical Module System for LF. In: Cheney, J., Felty, A. (eds.) Proceedings of the Workshop on Logical Frameworks: Meta-Theory and Practice (LFMTP), pp. 40–48. ACM Press, New York (2009)

[TB85] Trybulec, A., Blair, H.: Computer Assisted Reasoning with MIZAR. In: Joshi, A. (ed.) Proceedings of the 9th International Joint Conference on Artificial Intelligence, pp. 26–28 (1985)

[Wen10] Wenzel, M.: Asynchronous proof processing with Isabelle/Scala and Isabelle/jEdit. In: Sacerdoti Coen, C., Aspinall, D. (eds.) User Interfaces for Theorem Provers (UITP 2010). ENTCS. Elsevier, Amsterdam (2010)

[ZK09a] Zholudev, V., Kohlhase, M.: TNTBase: a Versioned Storage for XML. In: Proceedings of Balisage: The Markup Conference 2009. Balisage Series on Markup Technologies, vol. 3, Mulberry Technologies, Inc. (2009)

[ZK09b] Zholudev, V., Kohlhase, M.: TNTBase: a Versioned Storage for XML. In: Proceedings of Balisage: The Markup Conference 2009. Balisage Series on Markup Technologies, vol. 3, Mulberry Technologies, Inc. (2009)

Indexing and Searching Mathematics in Digital Libraries

Architecture, Design and Scalability Issues

Petr Sojka and Martin Líška

Masaryk University, Faculty of Informatics, Botanická 68a, 602 00 Brno, Czech Republic
sojka@fi.muni.cz, 255768@mail.muni.cz

Abstract. This paper surveys approaches and systems for searching mathematical formulae in mathematical corpora and on the web. The design and architecture of our MIaS (Math Indexer and Searcher) system is presented, and our design decisions are discussed in detail. An approach based on Presentation MathML using a similarity of math subformulae is suggested and verified by implementing it as a math-aware search engine based on the state-of-the-art system, Apache Lucene.

Scalability issues were checked based on 324,000 real scientific documents from arXiv archive with 112 million mathematical formulae. More than two billions MathML subformulae were indexed using our Solr-compatible Lucene extension.

Keywords: math indexing and retrieval, mathematical digital libraries, information systems, information retrieval, mathematical content search, document ranking of mathematical papers, math text mining, MIaS, WebMIaS.

I do not seek. I find.
Pablo Picasso

1 Introduction

The solution to the problem of mathematical formulae retrieval lies at the heart of building digital mathematical libraries (DML). There have been numerous attempts to solve this problem, but none have found widespread adoption and satisfaction within the wider mathematics community. And as yet, there is no widely accepted agreement on the math search format to be used for mathematical formulae by library systems or by Google Scholar.

MathML standard by W3C has become the standard for mathematics exchange between software tools. Almost no MathML is written directly by authors—they typically prefer a compact notation of some TEX flavour such as LATEX or AMS-LATEX. The designer of a search system for mathematics is thus faced with the task of converting data to a unifying format, and allowing DML users to use their prefered notation when posing queries. [AMS]LATEX or other TEX flavour are the typical preferences; Presentation MathML or Content MathML are used only when available as outputs of a software system.

J.H. Davenport et al. (Eds.): Calculemus/MKM 2011, LNAI 6824, pp. 228–243, 2011.

During the integration of existing DMLs into larger projects such as EuDML [15], the unsolved math search problem becomes evident—D*M*L without math search support is an oxymoron. As our subject matter search has not lead to a satisfactory solution, we have designed and implemented [7] *new* robust solutions for retrieval of mathematical formulae.

Section 2 explores published facts about research done in the area of mathematics retrieval. Pros and cons of existing approaches are outlined, most of them being neither applicable nor satisfactory for digital library deployment. In Section 3 we present our design of scalable and extensible system for searching mathematics, taking into account not only inherent structure of mathematical formulae but also formula unification and subformulae similarity measures. Our evaluation of prototypical implementation above the Apache Lucene open source full-featured search engine library is presented in Section 4. The paper closes listing future work directions in Section 5 and a conclusion is summarised in Section 6.

Computers are useless. They can only give you answers.
Pablo Picasso

2 Approaches to Searching Mathematics

A great deal of research on has been already undertaken on searching mathematical formulae in digital libraries and on the web. Several such Mathematical Search Engines (MSE) have been designed in the past: MathDex, EgoMath, LaTeXSearch, LeActiveMath or MathWebSearch. In this section, we will briefly comment on each of these.

MathDex[1] (formerly MathFind [9]) is a result of a NSF-funded project headed by Robert Miner of Design Science[2]. It encodes mathematics as text tokens, and uses Apache Lucene as if searching for text. Using similarity with search terms, ranked results are produced by the search algorithm, matching *n*-grams of presentation MathML. The creators of MathDex report that most of the work was due to a necessary and extensive normalization of MathML—because of the fact that it uses several converters and filters to convert to XHTML + MathML—HTML (jtidy), TeX/LaTeX (blahtex, LaTeXML, Hermes), Word (Word+MathType), PDF (pdf2tiff+Infty). The algorithm of *n*-gram ranking has several drawbacks. For one thing, it cannot take many kinds of elementary mathematical equivalences into account, and it puts undue weight on variable names.

Contrary to its intentions, MathDex has not become a sustainable service to the mathematical community, although it has fueled research in the area of mathematics searching [16,17,1].

EgoMath[3] is being developed by Josef Mišutka as an extension of a full text websearch core engine Egothor (by Leo Galamboš, MFF UK Prague) [8] licenced under GPL. It uses presentation MathML for indexing and develops generalization algorithms and

[1] www.mathdex.com/
[2] www.ima.umn.edu/2006-2007/SW12.8-9.06/activities/Miner-Robert/index.html
[3] egomath.projekty.ms.mff.cuni.cz/egomath/

relevancy calculation to cope with normalization. As part of EgoThor evaluation, an MSE evaluation dataset is also being developed[4].

LaTeXSearch[5] is a search tool offered by Springer in SpringerLink. It searches directly in the TeX math string representations as provided by the authors of papers submitted to Springer in LaTeX sources. Some kind of text similarity matching is probably used. Since it is not open source, one can only guess the strategy for posing queries. Our experiments typically lead to a very low precision. Neither is there any definition of the article dataset available.

LeActiveMath[6] search has been developed as part of the ActiveMath-EU project. It is Lucene based, indexing string tokens from OMDoc with an OpenMath semantic notation. The document database format is internal since only documents authored for LeActiveMath learning environments are indexed.

MathWebSearch[7] is an MSE developed in Bremen/Saarbrücken by Kohlhase et al. [2] It is not based on full text searching, rather it adopts a semantic approach: it uses substitution trees in memory. Both presentation and content MathML is supported, together with OpenMath. It is exceptional in the fact that it primarily deals with semantics and uses its own engine, not being built on the Lucene engine, for math. Further development is now being pursued under LaMaPun architecture [6].

The comparison of math search systems is summarized in Table 1. All of the MSEs reviewed had some drawbacks regarding their employment in a digital mathematical library such as EuDML. This was our main motivation for designing a new one, primarily for the use in large scale libraries, such as EuDML or ArXiv.

Everything you can imagine is real.
Pablo Picasso

3 Design of MIaS

We have developed a math-aware, full-text based search engine called *MIaS* (Math Indexer and Searcher). It processes documents containing mathematical notation in MathML format. MIaS allows users to search for mathematical formulae as well as the textual content of documents.

Since mathematical expressions are highly structured and have no canonical form, our system pre-processes formulae in several steps to facilitate a greater possibility of matching two equal expressions with different notation and/or non-equal, but similar formulae. With an analogy to natural language searching, MIaS searches not only for whole sentences (whole formulae), but also for single words and phrases (subformulae down to single variables, symbols, constants, etc.). For calculating the relevance of the matched expressions to the user's query, MIaS uses a heuristic weighting of indexed terms, which accordingly affects scores of matched documents and thus the order of results.

[4] egomath.cythres.cz/dataset.py
[5] www.latexsearch.com/
[6] devdemo.activemath.org/ActiveMath2/
[7] search.mathweb.org/index.xhtml

Table 1. Comparison of math search systems

System	Input documents	Internal representation	Approach	α-eq.	Query language	Queries	Indexing core
MathDex	HTML, TEX/LATEX, Word, PDF	Presentation MathML (as strings)	syntactic	✗	?	text, math, mixed	Apache Lucene
LeActiveMath	OMDoc, OpenMath	OpenMath (as string)	syntactic	✗	OpenMath (palette editor)	text, math, mixed	Apache Lucene
LATEXSearch	LATEX	LATEX (as string)	syntactic	✗	LATEX	titles, math, DOI	?
MathWeb Search	Presentation MathML, Content MathML, OpenMath	Content MathML, OpenMath (substitution trees)	semantic	✔	QMath, LATEX, Mathematica, Maxima, Maple, Yacas styles (palette editor)	text, math, mixed	Apache Lucene (for text only)
EgoMath	Presentation MathML, Content MathML, PDF	Presentation MathML trees (as strings)	mixed	✗	LATEX	text, math, mixed	EgoThor
MIaS	any (well-formed) MathML	Canonical Presentation MathML trees (as compacted strings)	math tree similarity/ normalization	✔	$\mathcal{AMS}$-LATEX or MathML	text, math, mixed	Apache Lucene/Solr

3.1 System Workflow

The top-level indexing scheme is shown in Figure 1. A detailed view of the mathematical part is shown in Figure 2 on the next page.

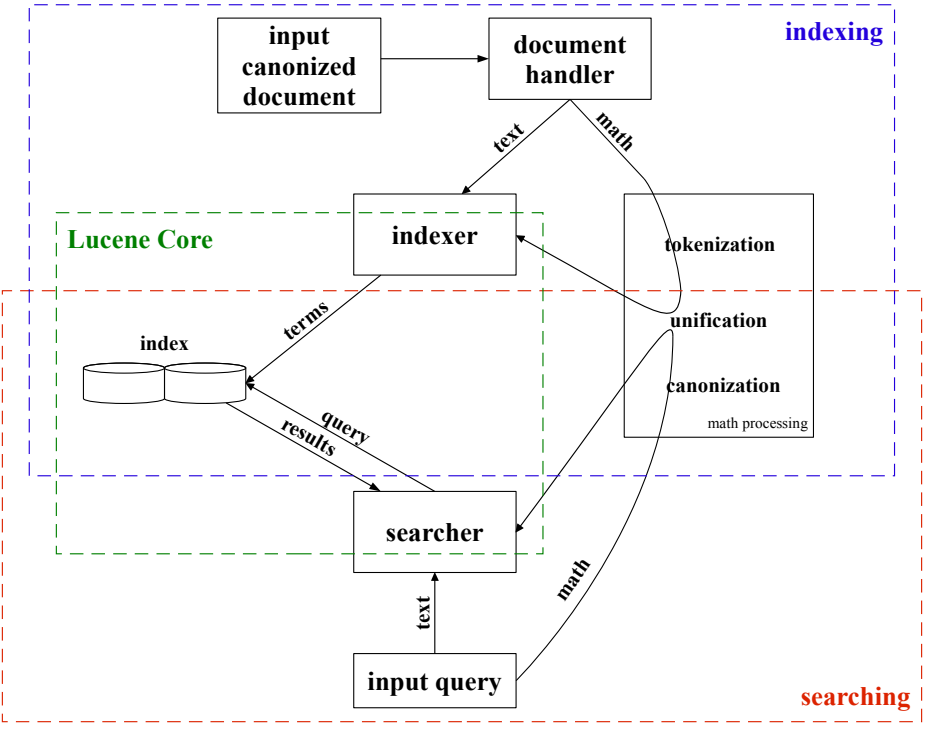


Fig. 1. Scheme of the system workflow

3.2 Indexing

MIaS is currently able to index documents in XHTML, HTML and TXT formats. As Figure 1 shows, the input document is first split into textual and mathematical parts. The textual content is indexed in a conventional way.

Mathematical expressions, on the other hand, are pre-analyzed in several steps to facilitate searches not only for exact whole formulae, but also for subparts (tokenization) and for similar expressions (formulae modifications). This addresses the issue of the static character of full-text search engines and creates several representations of each input formula all of which are indexed. Each indexed mathematical expression has a weight (relevancy score) assigned to it. It is computed throughout the whole indexing phase by individual processing steps following this basic rule of thumb—the more modified a formula and the lower the level of a subformula, the less weight is assigned to it.

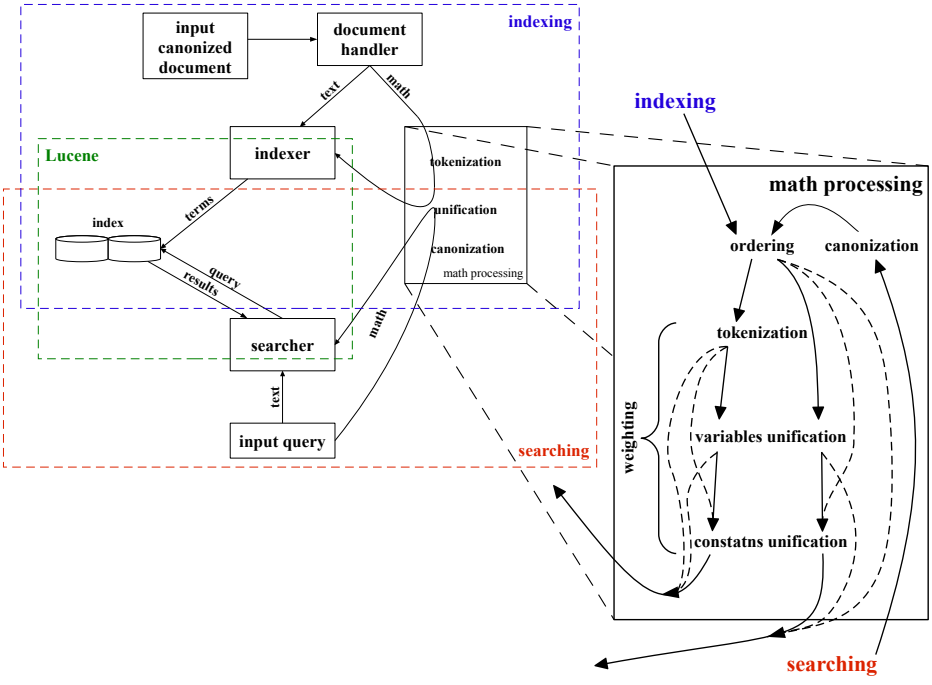


Fig. 2. Scheme of the MIaS workflow of math processing

At the end of all processing methods, formulae are converted from XML nodes to a compacted linear string form, which can be handled by the indexing core. Start and end XML tags are substituted by the tag name followed by an argument embraced by opening and closing parentheses. This creates abbreviated but still unambiguous representation of each XML node. For example, formula $a + b^2$, in MathML written as:

```
<math xmlns="http://www.w3.org/1998/Math/MathML">
  <mrow>
    <mi>a</mi>
    <mo>+</mo>
    <msup> <mi>b</mi><mn>2</mn></msup>
  </mrow>
</math>
```

is converted to "`math(mrow(mi(a)mo(+)msup(mi(b)mn(2))))`" and this string is then indexed by Lucene.

3.3 Tokenization

Tokenization is a straightforward process of obtaining subformulae from an input formula. MIaS makes use of Presentation MathML markup where all logical units are

enclosed in XML tags which makes obtaining all subformulae a question of tree traversal. The inner representation of each formula is an XML node encapsulating all the member child nodes. This means the highest level formula—as it appears in the input document—is represented by a node named "math". All logical subparts of an input formula are obtained and passed on to modification algorithms.

3.4 Formulae Modifications

MIaS performs three types of unification algorithms, the goal of which is to create several more or less generalized representations of all formulae obtained through the tokenization process. These steps allow the system to return similar matches to the user query while preserving the formula structure and α-equality.

3.5 Ordering

Let us take a simple example: $a + 3$ and the query $3 + a$. This would not match even though it is perfectly equal. This is why a simple ordering of the operands of the commutative operations, addition and multiplication, is used. It tries to order arguments of these operations in the alphabetical order of the XML nodes denoting the operands whenever possible—it considers the priority of other relevant operators in the formula. The system applies this function to the formula being indexed as well as to the query expression. Applied to the example above, the XML node denoting variable a is named "mi", the node denoting number 3 is named "mn". "mi"<"mn" therefore $3 + a$ would be exchanged for $a + 3$ and would match.

3.6 Unification of Variables

Let us take another example: $a + b^a$ and $x + y^x$. Again, these would not match even though the difference is only in the variables used. MIaS employs a process that unifies variables in expressions while taking bound variables into account. All variables are substituted for unified symbols (ids) in both the indexing and searching phases. Applied to the example, both expressions would unify to $\mathrm{id}_1 + \mathrm{id}_2^{\mathrm{id}_1}$ and would match. This process is not applied to single symbols—this would lead to the indexing of millions of ids and searching for any symbol would end up matching all of the documents containing it.

3.7 Unification of Constants

This is a strightforward process of substituting all the numerical constants for one unified symbol (const). This obviates the need for the exact values of constants in user queries. In some situations however, this can be too much of a generalization. As well as in the case of the variables, stand-alone numerical constants are not unified for the same obvious reason.

3.8 Formulae Weighting

During the searching phase, a query can match several terms in the index. However one match can be more important to the query than another, and the system must consider

this information when scoring matched documents. For mathematical formulae the system makes use of the processing operations described above since they all produce expressions more generalized than the input ones.

It is impossible to assemble a weighting function that is exactly right. Such a function should consider a document base on which the system will run as well as the established customs in a particular scientific field. We tried to create a complex and robust weighting function that would be appropriate to many fields.

The original unchanged untokenized formula should of course have the greatest weight, but the precision of the ordered representation is not compromised at all, so it should have the same weight. In fact, if the ordering process changes the order of some members in an expression, the original formula is not indexed at all. The starting weight for such a representation is 1.

The tokenization process should naturally lower the weight of the subformulae since they are deeper in the structure and therefore less important to the overall formula. When a user who is searching for $a + b$ finds two documents, the first containing $a + b$ and the second containing $\frac{2}{a+b}$, the first should score more and appear higher in the results, as it matches in higher *level* of MathML expression tree. Hence the tokenization process reduces the weight of the subformulae according to the level coefficient $l < 1$.

Both unification algorithms produce representations that are more generalized than their input expressions. They have a higher probability of matching, and should therefore score less. The unification of variables alters the weight of the result formula by coefficient $v < 1$, unification of number constants uses coefficient $c < 1$.

Theoretically, two equally unified subformulae matched on the same level of differently complex parent formulae would have the same score. For example

$$a + b^{3+a}$$

and

$$\frac{\int_0^\infty 25b^2\,db}{3+a} + \frac{d-e}{b} + 100ab$$

with the query $3 + a$. Both matches are not unified, and both are found on the third level. Analogously to conventional full-text engines which discriminate documents with more tokens than others, we use information about the complexity of parent formulae. More specifically, an initial weight of 1 is multiplied by the inverse number of nodes of a whole parent expression.

According to this model, each formula has a weight attribute indexed alongside itself, which belongs to the interval $(0, 1\rangle$. Weight w of the subformula contained on a certain level in a parent formula with the number of nodes (n) can be calculated in particular situations as follows:

- no changes made: $w = \frac{l^{\text{level}}(1+v+c+vc)}{n}$
- unified variables: $w = \frac{l^{\text{level}}(v+vc)}{n}$
- unified constants: $w = \frac{l^{\text{level}}(c+vc)}{n}$
- unified both variables and constants: $w = \frac{l^{\text{level}}(vc)}{n}$.

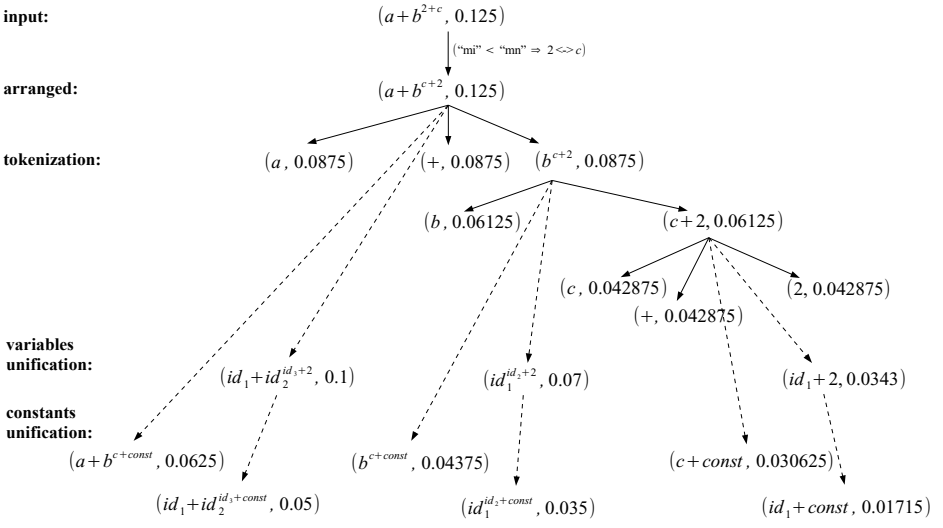


Fig. 3. Example of formula preprocessing. Ordered pairs are (<expression written naturally>, <it's weight>). All expressions as shown are indexed, except for the original one.

See Section 3.9 for details.

To fine tune the weighting parameters, we developed a tool with verbose output in which the behavior of the model can be observed and tested. A sample from the tool mentioned above is shown in Table 2 on the next page.

We have come to the conclusion that the unification of variables interferes less with original formula meaning than the unification of number constants. For this reason, its coefficient should be higher—i.e., less discriminating. The main question then became, how discriminating the level coefficient should be. Our empirical deduction is that going deeper in a structural tree should be discriminating, the precise match on a lower level should still score more than any unified formula on the level above, as could be seen in Table 2: $\frac{1}{a+3}$ (row 5) is an exact match on the second level and its score is higher than unified expressions matched on the first level (rows 2, 3 and 4).

This led us to the valuation of level weighting coefficient $l = 0.7$, unification weighting coefficient $v = 0.8$ and constant weighting coefficient $c = 0.5$.

In Figure 3 the whole formula preprocessing process is illustrated together with its subformulae weightings.

3.9 Searching

In the search phase, user input is again split into mathematical and textual parts. Formulae are then reprocessed in the same way as in the indexing phase, except for tokenization—which we doubt that users are likely to query, for example $\frac{a+b}{c}$ wanting to find documents only with occurrences of variable c. That means the queried

Table 2. Example of weighting function on several formulae. Original query is $a + 3$—all queried expressions are $a + 3$, $\mathrm{id}_1 + 3$, $a + \mathrm{const}$, $\mathrm{id}_1 + \mathrm{const}$.

Formula	Indexed Expressions	Score	Matched
$a + 3$	$0.25=[a + 3]$, $0.2=[\mathrm{id}_1 + 3]$, $0.175=[a, 3, +]$, $0.125=[a + \mathrm{const}]$, $0.1=[\mathrm{id}_1 + \mathrm{const}]$	2.7	$0.1[\mathrm{id}_1 + \mathrm{const}] + 0.25[a + 3] + 0.2[\mathrm{id}_1 + 3] + 0.125[a + \mathrm{const}]$
$b + 3$	$0.25=[b + 3]$, $0.2=[\mathrm{id}_1 + 3]$, $0.175=[b, +, 3]$, $0.125=[b + \mathrm{const}]$, $0.1=[\mathrm{id}_1 + \mathrm{const}]$	1.2	$0.1[\mathrm{id}_1 + \mathrm{const}] + 0.2[\mathrm{id}_1 + 3]$
$a + 5$	$0.25=[a + 5]$, $0.2=[\mathrm{id}_1 + 5]$, $0.175=[a, +, 5]$, $0.125=[a + \mathrm{const}]$, $0.1=[\mathrm{id}_1 + \mathrm{const}]$	0.9	$0.1[\mathrm{id}_1 + \mathrm{const}] + 0.125[a + \mathrm{const}]$
$c + 10$	$0.25=[c + 10]$, $0.2=[\mathrm{id}_1 + 10]$, $0.175=[c, +, 10]$, $0.125=[c + \mathrm{const}]$, $0.1=[\mathrm{id}_1 + \mathrm{const}]$	0.4	$0.1[\mathrm{id}_1 + \mathrm{const}]$
$\frac{1}{a+3}$	$0.16667=[\frac{1}{a+3}]$, $0.13334=[\frac{1}{\mathrm{id}_1+3}]$, $0.11667=[1, a + 3]$, $0.09334=[\mathrm{id}_1 + 3]$, $0.08334=[\frac{\mathrm{const}}{a+\mathrm{const}}]$, $0.08167=[+, 3, a]$, $0.06667=[\frac{\mathrm{const}}{\mathrm{id}_1+\mathrm{const}}]$, $0.05833=[a + \mathrm{const}]$, $0.04667=[\mathrm{id}_1 + \mathrm{const}]$	1.26	$0.04667[\mathrm{id}_1 + \mathrm{const}] + 0.11667[a + 3] + 0.09334[\mathrm{id}_1 + 3] + 0.05833[a + \mathrm{const}]$
$\frac{1}{b+3}$	$0.16667=[\frac{1}{b+3}]$, $0.13334=[\frac{1}{\mathrm{id}_1+3}]$, $0.11667=[b + 3, 1]$, $0.09334=[\mathrm{id}_1 + 3]$, $0.08334=[\frac{\mathrm{const}}{b+\mathrm{const}}]$, $0.08167=[b, 3, +]$, $0.06667=[\frac{\mathrm{const}}{\mathrm{id}_1+\mathrm{const}}]$, $0.05833=[b + \mathrm{const}]$, $0.04667=[\mathrm{id}_1 + \mathrm{const}]$	0.56	$0.04667[\mathrm{id}_1 + \mathrm{const}] + 0.09334[\mathrm{id}_1 + 3]$
$\frac{1}{a+5}$	$0.16667=[\frac{1}{a+5}]$, $0.13334=[\frac{1}{\mathrm{id}_1+5}]$, $0.11667=[1, a + 5]$, $0.09334=[\mathrm{id}_1 + 5]$, $0.08334=[\frac{\mathrm{const}}{a+\mathrm{const}}]$, $0.08167=[a, 5, +]$, $0.06667=[\frac{\mathrm{const}}{\mathrm{id}_1+\mathrm{const}}]$, $0.05833=[a + \mathrm{const}]$, $0.04667=[\mathrm{id}_1 + \mathrm{const}]$	0.42	$0.04667[\mathrm{id}_1 + \mathrm{const}] + 0.05833[a + \mathrm{const}]$
$\frac{1}{c+10}$	$0.16667=[\frac{1}{c+10}]$, $0.13334=[\frac{1}{\mathrm{id}_1+10}]$, $0.11667=[1, c + 10]$, $0.09334=[\mathrm{id}_1 + 10]$, $0.08334=[\frac{\mathrm{const}}{c+\mathrm{const}}]$, $0.08167=[+, c, 10]$, $0.06667=[\frac{\mathrm{const}}{\mathrm{id}_1+\mathrm{const}}]$, $0.05833=[c + \mathrm{const}]$, $0.04667=[\mathrm{id}_1 + \mathrm{const}]$	0.19	$0.04667[\mathrm{id}_1 + \mathrm{const}]$

expressions are first ordered, then unified. This produces several representations which are connected to the final query by the logical OR operator.

Textual query terms are connected to the final query by the logical AND operator. Therefore by specifying a text term we can narrow down the results, because each returned document must have the term contained. When more than one text term is specified, they are implicitly connected to the text query by the OR operator which means at least one term should occur in the result. We can also explicitly state preferences about each text term—whether it needs to occur in the result or not.

As stated above, the final query, without having explicitly stated occurrences of text terms, is in the logical form of $(\mathrm{formula}_1 \vee \ldots \vee \mathrm{formula}_n) \wedge (\mathrm{term}_1 \vee \ldots \vee \mathrm{term}_n)$.

In order to counterbalance the weight of the textual and mathematical parts of the query, the score of the matched formulae are additionally multiplied by number of nodes

the matching query consists of. This results in more complex mathematical queries scoring more.

A very positive value has its price in negative terms...
the genius of Einstein leads to Hiroshima.
Pablo Picasso

4 Evaluation

For large scale evaluation, we needed an experimental implementation and a corpus of mathematical texts.

4.1 Implementation

The Math Indexer and Searcher is written in Java. The role of full-text indexing and searching core is performed by Apache Lucene 3.1.0. The mathematical part of document processing can be seen as a standalone pluggable extension to any full-text library, however it would need custom integration for each one. In the case of Lucene, a custom Tokenizer (MathTokenizer) has been implemented.

For the textual content of documents, Lucene's StandardAnalyzer is employed. In MathTokenizer, TermAttributes are used for carrying strings of math expressions and PayloadAttribute for storing weights of formulae.

The question now is, how should the weights of formulae be taken into consideration in the overall score of matched documents. Lucene's practical scoring function for every hit document d by query q with each query term t is as follows:

$$score(q,d) = coord(q,d) \cdot queryNorm(q) \cdot \sum_{t\ in\ q} \left(tf(t \text{ in } d) \cdot idf(t)^2 \cdot t.getBoost() \cdot norm(t,d) \right)$$

It is described in detail at `http://lucene.apache.org/java/3_1_0/api/core/index.html?org/apache/lucene/search/Similarity.html`.

When searching for mathematical formulae, their weights need to be considered in the final score of the document. The resulting MIaS scoring function adds another parameter to the basic function—weight w of one matched formula:

$$score(q,d) = coord(q,d) \cdot queryNorm(q) \cdot \cdot \sum_{t\ in\ q} \left(tf(t \text{ in } d) \cdot avg(w) \cdot idf(t)^2 \cdot t.getBoost() \cdot norm(t,d) \right) \quad (1)$$

If a document contains the same formula more than once (each occurrence can have different weight assigned), the average value of all the weights is taken into consideration ($avg(w)$).

Let's take a simplified version of the function (1). Specifically, let us not to consider normalization factor $queryNorm(q)$, inverse document frequency $idf(t)^2$ and document/field boost and length factor $norm(t,d)$:

$$score(q,d) = coord(q,d) \cdot \sum_{t\ in\ q} (tf(t \text{ in } d) \cdot avg(w) \cdot t.getBoost()) \quad (2)$$

and follow the example in Table 2 on page 237. Let's consider we query a document containing only two formulae $b + 3$ and $\frac{1}{a+3}$ (rows 2 and 5). During indexing time, preprocessing creates several more representations, all of which are indexed (shown in the second column). The query is $a + 3$ which is expanded by query preprocessing to the final query that takes the form of $a + 3 \vee \mathrm{id}_1 + 3 \vee a + \mathrm{const} \vee \mathrm{id}_1 + \mathrm{const}$. Column 4 shows which actual expressions will match the query for each particular input formula. $coord(q, d)$ will be $\frac{4}{4}$ because all four of the four query terms found a match. Query terms $a + 3$ matched only one indexed term and its weight is 0.11667; query term $a + \mathrm{const}$ also matched only one indexed term and its weight 0.05833; query term $\mathrm{id}_1 + 3$ matched two indexed terms with weights 0.2 and 0.09334 so its average is 0.14667; finally the last query term $\mathrm{id}_1 + \mathrm{const}$ matched two indexed expressions with weights 0.1 and 0.04667 so its average is 0.07335. *t.getBoost*() is a query time boosting factor and as stated in Section 3.9, we use the number of XML nodes of the original query formula—in this example it is 4. The resulting score of the whole document is then

$$\frac{4}{4} \cdot \left((1 \cdot 0.11667 \cdot 4) + (1 \cdot 0.05833 \cdot 4) + (2^{\frac{1}{2}} \cdot 0.14667 \cdot 4) + (2^{\frac{1}{2}} \cdot 0.07335 \cdot 4)\right)$$

4.2 Corpus of Mathematical Documents MREC

A document corpus MREC with 324,060 scientific documents (version 2011.3.324) was initially used to evaluate the behaviour of the system we modelled. The documents come from the arXMLiv project that is converting document sets from arXiv into XHTML + MathML (both Content and Presentation) [13]. At the time of testing, our system was not yet able to process mixed MathML markup so preprocessing in the sense of filtering out unwanted markup was needed. The resulting corpus size was 53 GB uncompressed, 6.7 GB compressed. Documents contained 112,055,559 formulae in total, of which 2,129,261,646 mathematical expressions were indexed. The resulting index size was approx. 45 GB.

We were able to gather even greater amount of documents in MREC corpus version 2011.4.439 to test our indexing system. This corpus consists of 439,423 arXMLiv documents containing 158,106,118 mathematical formulae. 2,910,314,146 expressions were indexed and the resulting size of the index is 63 GB. Sizes of uncompressed and compressed corpora size are 124 GB and 15 GB, respectively.

Mentioned MREC corpora are available to the community for download from MREC web page `http://nlp.fi.muni.cz/projekty/eudml/MREC/` so that other math indexing engines could be compared with MIaS on the same data.

4.3 Results

Math Indexer and Searcher demonstrated the ability to index and search a relatively vast corpus of real scientific documents. Its usability is highly elevated thanks to its preprocessing functions together with formulae weighting model. The ability to search for exact and similar formulae and subformulae, more so with customizable relevancy computation, demonstrates an unquestionable contribution to the whole search experience.

Table 3. Scalability test results (run on 32 GB RAM, quad core AMD Opteron™ Processor 850 driven machine)

Documents	Input formulae	Indexed formulae	Indexing time [min]	Average query time [ms]
10,000	3,450,114	65,194,737	39.15	32
50,000	17,734,342	334,078,835	201.68	178
200,000	70,102,960	1,316,603,055	889.28	576
324,060	112,055,559	2,129,261,646	1,292.16	789

It is very difficult, if not impossible, to completely verify the correctness of the theoretical considerations made in the previous sections and thus correctness of search results. For this purpose, a sufficiently large corpus of documents with fully controlled content would be needed. For any assembled query, there should exist beforehand a complete list of the documents ordered by their relevance to the query to compare the actual results to.

We have applied an empirical approach to the evaluation so far. For these purposes we have created a demo web interface WebMIaS which is publicly available on the MIaS web page `http://nlp.fi.muni.cz/projekty/eudml/mias/`. It works over MREC corpora discussed in the Section 4.2. Additionally, for the latest MREC corpus we have implemented and added demanded snippet generation and mathematical match highlighting in hit list. Preliminary version of this functionality is available.

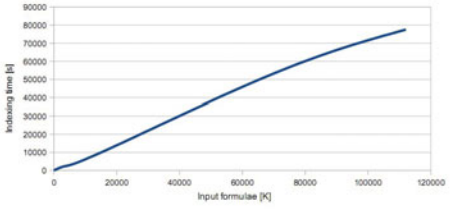
Our WebMIaS interface supports queries in two different notations—in $\mathcal{AMS}$-LaTeX and MathML. Mathematical queries are additionally canonized using XSLT transformations from UMCL library [4.3] to improve the query and to avoid notation flaws restraining proper results retrieval. Portability of the interface is increased by using MathJax for rendering of mathematical formulae in snippets.

4.4 Scalability Testing and Efficiency

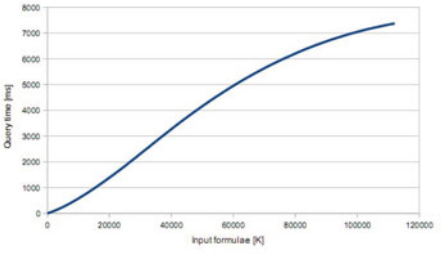
We have devised a scalability test to see how the system behaves with an increasing number of documents and formulae indexed. Subsets containing 10,000, 50,000, 200,000 and the complete 324,060 documents were gradually indexed and several values were measured: the number of input formulae, the number of indexed formulae, the indexing time and the average query time.

The number of input formulae indicate how rich a particular subset was in formulae; the number of indexed formulae should illustrate their complexity. Moreover both should indicate whether indexing and query time are dependent on the number of documents or specially on the formulae they contain. For measuring the average query time, we queried each created index with the same set of differently complex queries (mixed, non-mixed, more/less complex single/multiple formulae) computing the average time. The results are shown in Table 3 and in the form of diagrams in Figure 4 on the next page.

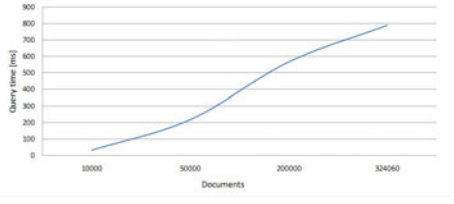
MREC version 2011.4.439 was indexed using improved and optimised algorithms and ran on a different machine. Therefore it cannot be compared to measured values



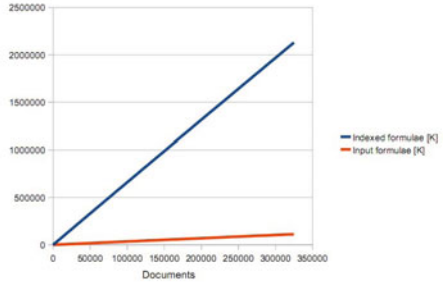
(a) Number of input formulae vs. indexing time



(b) Number of input formulae vs. query time



(c) Number of documents vs. query time



(d) Number of formulae vs. number of documents

Fig. 4. Scalability diagrams

shown in tables 3 on the facing page and 4. Indexing time of this corpus was 1378.82 ms, e.g. almost 23 hours.

He can who thinks he can, and he can't who thinks he can't.
This is an inexorable, indisputable law.
Pablo Picasso

5 Open Issues, Future Work

We are now awaiting heterogeneous MathML data collected by the EuDML project, that has been generated from born-digital [meta]data [10], from born-digital PDFs [5] or from math OCR [14].

It is evident that some kind of normalization of MathML will be a necessity. We have opted for Canonical MathML [4,3] as normalization MathML format and are using software library UMCL supporting it. Our latest experiments with canonical form of MathML generated by UMCL shows that it not only increases fairness of similarity ranking, but also helps to match a query against the indexed form of MathML. We are also working hard on snippets generation and on matched formulae visualization.

Another area of long-term research planned is supporting Content MathML, in a way similar to the current handling of Presentation MathML. The architectural design

is open to it, but as most of math within EuDML will be in Presentation MathML taken from PDFs, this is not currently a high priority.

> I am always doing that which I can not do, in order that I may learn how to do it.
> Pablo Picasso

6 Conclusions

We have presented an approach to mathematics searching and indexing—the architecture and design of the MIaS system. The feasibility of our approach has been verified on large corpora of real mathematical papers from arXMLiv. Scalability tests have confirmed that the computing power needed for fine math similarity computations is readily available; this would allow the use of this technology for projects on a European or world-wide scale.

Acknowledgements. This work has been in part financed by the European Union through its Competitiveness and Innovation Programme (Information and Communications Technologies Policy Support Programme, "Open access to scientific information", Grant Agreement no. 250,503). We thank anonymous reviewers for their improvement and future work suggestions, Michal Růžička for help with figure drawings and web form of MIaS interface, and Peter Mravec for collecting MREC.

References

1. Altamimi, M., Youssef, A.S.: A Math Query Language with an Expanded Set of Wildcards. Mathematics in Computer Science 2, 305–331 (2008), http://dx.doi.org/10.1007/s11786-008-0056-4
2. Anca, Ş.: Natural Language and Mathematics Processing for Applicable Theorem Search. Master's thesis, Jacobs University, Bremen (August 2009), https://svn.eecs.jacobs-university.de/svn/eecs/archive/msc-2009/aanca.pdf
3. Archambault, D., Berger, F., Moço, V.: Overview of the "Universal Maths Conversion Library". In: Pruski, A., Knops, H. (eds.) Assistive Technology: From Virtuality to Reality: Proceedings of 8th European Conference for the Advancement of Assistive Technology in Europe AAATE 2005, Lille, France, pp. 256–260. IOS Press, Amsterdam (September 2005)
4. Archambault, D., Moço, V.: Canonical MathML to Simplify Conversion of MathML to Braille Mathematical Notations. In: Miesenberger, K., Klaus, J., Zagler, W., Karshmer, A. (eds.) ICCHP 2006. LNCS, vol. 4061, pp. 1191–1198. Springer, Heidelberg (2006), http://dx.doi.org/10.1007/11788713_172
5. Baker, J.B., Sexton, A.P., Sorge, V.: Extracting Precise Data on the Mathematical Content of PDF Documents. In: Sojka [11], pp. 75–79, http://dml.cz/handle/10338.dmlcz/702535
6. Grigore, M., Wolska, M., Kohlhase, M.: Towards context-based disambiguation of mathematical expressions. Math-for-Industry Lecture Note Series, vol. 22, pp. 262–271 (December 2009)
7. Líška, M.: Vyhledávání v matematickém textu (in Slovak), Searching Mathematical Texts. Bachelor Thesis, Masaryk University, Brno, Faculty of Informatics (advisor: Petr Sojka) (2010), https://is.muni.cz/th/255768/fi_b/?lang=en

8. Mišutka, J., Galamboš, L.: Extending Full Text Search Engine for Mathematical Content. In: Sojka [11], pp. 55–67, http://dml.cz/dmlcz/702546
9. Munavalli, R., Miner, R.: MathFind: A Math-Aware Search Engine. In: Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2006, pp. 735–735. ACM, New York (2006), http://doi.acm.org/10.1145/1148170.1148348
10. Růžička, M., Sojka, P.: Data Enhancements in a Digital Mathematics Library. In: Sojka [12], pp. 69–76, http://dml.cz/dmlcz/702575
11. Sojka, P. (ed.) Towards a Digital Mathematics Library, Birmingham, UK. Masaryk University (July 2008), http://www.fi.muni.cz/~sojka/dml-2008-program.xhtml
12. Sojka, P. (ed.) Towards a Digital Mathematics Library, Paris, France. Masaryk University (July 2010), http://www.fi.muni.cz/~sojka/dml-2010-program.html
13. Stamerjohanns, H., Kohlhase, M., Ginev, D., David, C., Miller, B.: Transforming Large Collections of Scientific Publications to XML. Mathematics in Computer Science 3, 299–307 (2010), http://dx.doi.org/10.1007/s11786-010-0024-7
14. Suzuki, M., Tamari, F., Fukuda, R., Uchida, S., Kanahori, T.: INFTY — An integrated OCR system for mathematical documents. In: Vanoirbeek, C., Roisin, C., Munson, E. (eds.) Proceedings of ACM Symposium on Document Engineering 2003, Grenoble, France, pp. 95–104. ACM, New York (2003)
15. Sylwestrzak, W., Borbinha, J., Bouche, T., Nowiński, A., Sojka, P.: EuDML—Towards the European Digital Mathematics Library. In: Sojka [12], pp. 11–24, http://dml.cz/dmlcz/702569
16. Youssef, A.S.: Roles of Math Search in Mathematics. In: Borwein, J., Farmer, W. (eds.) MKM 2006. LNCS (LNAI), vol. 4108, pp. 2–16. Springer, Heidelberg (2006), http://dx.doi.org/10.1007/11812289_2
17. Youssef, A.S.: Methods of Relevance Ranking and Hit-Content Generation in Math Search. In: Kauers, M., Kerber, M., Miner, R., Windsteiger, W. (eds.) MKM/CALCULEMUS 2007. LNCS (LNAI), vol. 4573, pp. 393–406. Springer, Heidelberg (2007)

Isabelle as Document-Oriented Proof Assistant

Makarius Wenzel

Université Paris-Sud 11, LRI
Orsay, France
http://www.lri.fr/ wenzel/

Abstract. Proof assistants in the LCF tradition, such as Coq, Isabelle, and the HOL family, are notorious for old-fashioned command-line interaction with input and output of plain text. Established prover interfaces like Proof General merely add a thin layer on top of the read-eval-print loop in the background. More sophisticated mathematical editors, Web-services, Wiki-servers for mathematical content do exist, but any project that aims at fully formal proof-checking struggles with recurrent problems posed by ancient prover engines.

Taking the perspective of Isabelle, we discuss common problems and solutions that have emerged in the past few years, to fit the prover smoothly into a document-oriented environment with rich semantic annotations for formal sources. For example, this enables a conventional editor framework to present formal content provided by the prover, without having to understand logic itself (or re-implement a prover). This can be achieved with minimal changes on the editor and prover side, but the combination is able to support the usual metaphors of squiggly underline, tooltips, popups etc. that are now commonplace in browsers or IDEs.

Many of these document-oriented traits of current Isabelle are sufficiently general to be transferred to other provers. If such principles are becoming routinely available in LCF-style provers, building combined mathematical assistants should become more feasible.

1 Introduction

Isabelle has been centered around some notion of *formal proof document* ever since the introduction of the Isar proof language around 1999 (see also the overview [12]). This philosophical notion of "document" emphasizes an idea of *primary presentation* of machine-checked proofs in human-readable form. To accompany the abstract principles with concrete implementations, Isabelle/Isar provides some means to produce pretty-printed versions based on superficial observations on the structure of source text that is known to have passed through formal checking. Thus the traditional Isabelle document preparation system can present theories with good typesetting quality in PDF-LaTeX, such as the present paper. Little of the formal content that the prover has accumulated in checking the text is available in the resulting PDF, though.

In the past few years, Isabelle has started to provide more substantial access to internal prover content. This was motivated by the requirements of formal

J.H. Davenport et al. (Eds.): Calculemus/MKM 2011, LNAI 6824, pp. 244–259, 2011.

theory repositories and the user agents to access the content (via Web-interfaces, Wikis, Prover IDEs). The general question is how aspects of sophisticated formal processing of sources can be represented externally, without front-ends having to understand logic or re-implement provers, and without provers having to implement their own front-end technology. Some of this advanced document-oriented functionality of Isabelle is already accessible to users in the experimental Isabelle/jEdit Prover IDE [13]: results of proof checking are visualized as squiggly underlines, tooltips, or hyperlinks in the source. This is only the beginning of many more possibilities, based on the same document-oriented architecture.

These new concepts of Isabelle are sufficiently general to be applicable to other proof assistants, such as Coq or variants of HOL. Although not a prover, the former command-line compiler of Poly/ML (by David Matthews) has already been integrated: Isabelle/ML sources that are embedded into the formal context benefit from content revealed by Isabelle and the static analysis of SML. This makes Isabelle/jEdit also an IDE for Poly/ML.

Cultural Side-conditions and Programming Paradigms. We need to respect some cultural side-conditions, if old-school provers shall become native participants in a larger world that typically speaks XML instead of λ-calculus.

Major interactive theorem provers are implemented in some functional programming language. For systems from North America this usually means LISP, e.g. see ACL2 [16, §8] or PVS [16, §3]. For European systems it often means a descendant of ML [6]: Standard ML for Isabelle [16, §6] and some HOL variants, OCaml for Coq [16, §4] and other HOL variants [16, §1], or Haskell for some newer implementations such as Agda [16, §7].

This cultural background impacts our problem of document-oriented theorem provers. Implementors of proof assistants routinely use higher-order datatypes and Hindley-Milner polymorphism, but the outside world might only speak of XML and event-dispatch objects for SAX parsing. The Haskell community has managed to adopt significant parts of the XML world as libraries like HaXml, but less is available in ML. Our general attitude is to retain the established prover programming culture of ML, and extend our repertoire by a few elements that help to integrate smoothly with XML-based document markup. In Isabelle, the general prover integration problem is further materialized by the following bilingual approach:

Isabelle/ML represents the pure symbolic programming environment for sophisticated logical concepts implemented in the prover. Isabelle happens to use Standard ML (notably the Poly/ML implementation), which is also embedded into the logical context [15].

Isabelle/Scala provides an external API for prover integration on the Java VM, where many useful IDE or Web frameworks already exist. Scala/JVM [7] is sufficiently flexible to support the received ML programming style. Sometimes it goes beyond that, e.g. via typed views on untyped data.

Our document-oriented concepts are reflected both in Isabelle/ML and Isabelle/Scala. The main ideas are already present in ML, and can be re-used

without subscribing to the bilingual architecture. Conceptually, our Scala library provides a simplified programming model, without exposing the full details of document-oriented prover protocols to front-ends as was done in PGIP [4].

Scala is able to express the sublime programming style of Isabelle/ML, with parameterized types, higher-order combinators, pattern matching etc. It also provides direct access to existing Java frameworks at the JVM bytecode level. Although it is hard to imagine sophisticated provers implemented in Java, one might seriously consider doing it in Scala. For us the latter is a theoretical question, since we aim at re-use of provers in ML, not re-writing from scratch in Scala. There are also some inherent limitations of the JVM that make it hard to work with large symbolic structures as effectively as in existing ML implementations.

Overview. This paper is structured as follows. §2 introduces principles for robust support of text that is annotated by markup. This makes the prover speak XML natively with minimal impact on the existing code base. §3 presents some common uses of XML content in the prover for data representation and physical rendering. §4 describes a general scheme that can reconstruct content of the conceptual prover document model in external form.

2 Text with Markup

Proof assistants are traditionally based on text for input and output. We remain faithful to this principle, but add markup information in a conservative manner. Making this work smoothly for existing provers turns out to be surprisingly difficult: it might explain retrospectively why this is not yet established prover technology. We first need to answer questions like "What is text?" and "What is markup?" precisely, and keep in mind various fine points for our implementation.

2.1 Text Encoding and Rendering

Proof assistants have a natural demand for mathematical symbols, beyond traditional ASCII art. In recent years, Unicode has become sufficiently wide-spread to be considered, although it is not as universal and uniform as ASCII. Unicode standards are still evolving, and even within a given version there are variations.

Unicode defines tables of *codepoints* (represented as small integers) that are associated with names and visual appearance of glyphs. Physical representation in memory is defined separately by *encodings*: most common are now UTF-8 and UTF-16, both available in several variants. Thus the correspondence of bytes in memory to glyphs is not immediately obvious, and moving between different platforms such as ML (UTF-8) and JVM (UTF-16) requires some care.

For example, $\mathcal{A}$ (codepoint `U+1D49C`, `MATHEMATICAL SCRIPT CAPITAL A`) corresponds to 4 bytes in UTF-8 (due to its regular multi-byte encoding), and 4 different bytes in UTF-16 (due to "surrogate characters" that are used to work

around limitations of classic 16 bit Unicode). Some intermediate program module that is unaware of UTF-16 surrogates might recode the two wide characters behind $\mathcal{A}$ separately and arrive at yet another (wrong) byte sequence.

This illustrates some subtleties that need to be taken into account for robust representation of mathematical source text inside our provers. Even though the "supplementary codepoints" outside the 16 bit range provide interesting mathematical symbols and mathematical alphabets in the spirit of TEX math mode (variations of latin, greek, calligraphic, bold, italics), we need to be careful about depending on such corner cases of the Unicode standards.

Further uncertainty is caused by semantic alternatives in the collection of codepoints. For example, the visual appearance of ϕ (`\phi`) vs. φ (`\varphi`) in TEX is defined by Knuth's Metafonts, but Unicode symbol fonts often disagree. The official entry[1] for codepoint `U+03C6` says "the ordinary Greek letter, showing considerable glyph variation", and refers to the alternative codepoint `U+03D5` among others. Taking mathematical typesetting seriously, we should probably use codepoint `U+1D711 MATHEMATICAL ITALIC SMALL PHI` to imitate math italics in TEX, but the front-end would need a font that implements that glyph[2].

Problems

- Robust encoding of text that is suitable for long-term storage (avoiding recoding every few years to accommodate new Unicode standards).
- Robust rendering of symbols, potentially depending on the capabilities of the front-end encoding (e.g. 16 bit limit) and fonts (e.g. missing glyphs).

Solutions: We propose variants (a), (b), (c) as follows.

(a) Trusting that UTF-8 actually manages to represent Unicode text adequately in the next decades, prover sources are directly encoded using this particular standard. Semantic dilemmas in the official Unicode tables are resolved by our own internal standardization, judging glyph variants wrt. availability in important font families (such as DejaVu or STIX) and selecting codes for φ etc. once and for all.

(b) We are conservative about ASCII as main physical representation of text, but add explicitly named entities in the spirit of TEX. Already since 1998, Isabelle has supported the notation `\<`*name*`>` for that purpose, where *name* follows usual alphabetic identifier syntax. This provides an *infinite* collection of non-ASCII symbols that is *independent* of encoding.

Extra tables define the rendering of a finite subset of Isabelle symbols. For example, `\<forall>` can be mapped to `\forall` in TEX and `U+2200` in Unicode. Such tables can be fine-tuned over time to adjust to coming

[1] http://www.unicode.org/charts/PDF/U0370.pdf

[2] After many years of delay, the STIX project http://www.stixfonts.org has managed to release version 1.0 of its free mathematical fonts in 2010. So fairly complete coverage of mathematical glyphs may be become more commonplace in the near future, although its rendering quality for screen resolution is not ideal.

trends in Unicode and availability of mathematical fonts, without changing the main body of formal sources. Users can augment the tables by domain-specific notation for their own applications.

(c) Careful combination of (a) + (b). The formal language prefers named symbols of the form \<*name*> but the user is allowed to refer to UTF-8 Unicode, say in informal text. Since UTF-8 is conservative over 7-bit ASCII byte streams anyway, this hardly requires any change of the prover. Even though the prover never uses genuine Unicode in the standard libraries, users can write text in foreign scripts like Cyrillic or Chinese.

Although the re-mapping of (b) introduces some technical inconveniences in the implementation, such adjustments have been part of Proof General [3] for many years, re-using older packages for TEX. The JVM also allows to register character encodings defined in user space. The combination of (c) is already officially recognized in Isabelle2011 [14, §1.2.1]: it mainly refines the 1998 version of (b) by precise of addressing Unicode text positions as explained in §2.2.

In summary, we essentially use Unicode for *poor man's rendering* of mathematical text, using those parts of the technology that are known to work reliably.

2.2 Markup via Text Addressing

We define *markup* as any information that is adjoined with the original source text. In XML the markup elements happen to consist of a *name* and *property list* (called *attributes* in XML terminology). We take care to coincide with such public standards when delivering documents to external tools, although internally the exact structure of markup does not matter. In Isabelle/Scala many operations on markup are either parameterized over some arbitrary type `A` or use the universal union type `Any`.

There are two interchangeable ways to represent text together with nested markup:

1. Explicit trees that alternate markup nodes with body text (in the spirit of XML trees, see also §3).
2. Implicit annotations for the text as separate mapping from text intervals to markup nodes (maintained as side-result of formal checking, see also §4).

The second variant is not immediately obvious, but turns out very convenient for the prover in most situations: it can continue to digest text as before, but occasionally reports annotations about its internal state, in correspondence to precise source positions. So the markup problem is reduced to robust addressing of text positions, to anchor the attached semantic information in its proper place (e.g. inferred type information, or warnings and errors that refer to parts of text).

Problem: Robust physical addressing of text offsets that is suitable for persistent storage and stable under change of encodings or rendering (cf. §2.1).

Solution: Given a byte stream, individual *symbols* are identified as follows:

- a single ASCII character (byte 0...127)
- a codepoint according to UTF-8 (byte 128...255 followed by the longest possible sequence of bytes 128...191)
- a named symbol `\<`*name*`>` where *name* consists of ASCII letters and digits
- a malformed symbol `\<` that cannot be extended to a named symbol

This defines a *total* scan function on packed symbol sequences, which is available as `Symbol.explode: string -> string list` in Isabelle/ML. Totality is achieved by tolerating malformed symbols (including illegal UTF-8, which happens to be self-synchronizing). Further syntax layers can reject those, e.g. in lexical analysis. Positions gained from counting symbols are passed through the syntax hierarchy as intervals of logical offsets. Raw byte addressing is not used.

Isabelle/Scala is subject to UTF-16 on the JVM, but the same notion of symbols can be accommodated. We provide some *iterator* in Scala that enumerates the symbols as consecutive intervals over JVM strings. The handling of surrogate UTF-16 characters is also included here, to ensure that logical codepoints are counted in the same way as the corresponding UTF-8 sequences in Isabelle/ML.

2.3 YXML Transfer Syntax

Proof assistants prefer to output plain text, but we would like to augment that by markup information. It is unrealistic to expect that existing ML code for thousands of messages in the prover and add-on tools are reworked to use XML trees instead of strings. Here is a typical example from the Isabelle sources, which composes literal text and formatted output of pretty-printed formal entities:

```
error ("Unbound schematic variable: " ^ Syntax.string_of_term ctxt t)
```

Error messages are particularly delicate, since they only occur in exceptional situations. It would take a long time to discover the mistakes that are inevitably introduced by forcing XML onto the prover in a crude manner.

Problem: Robust presentation of markup for plain text, such that markup is orthogonal to the text (with its encoding of symbols, cf. §2.1) and markup can be nested safely (applied to text that might have been marked before).

Solution: First we observe that text produced by the prover never contains low-ASCII control characters, except for white-space. Second we observe exactly the same for regular text characters in the XML standard[3]. Third we check old ASCII control tables, and choose $\mathbf{X} = chr\ 5$ and $\mathbf{Y} = chr\ 6$, which are unlikely to have any special meaning in current computer systems. We can now define our own *YXML transfer syntax* for XML documents as follows:

	XML	YXML
open tag	`<`*name attribute*`=`*value ...*`>`	**XY***name*(**Y***attribute*`=`*value*...)**X**
close tag	`</`*name*`>`	**XYX**

[3] http://www.w3.org/TR/xml/#charsets

Note that there is no special case for elements with empty body text, i.e. `<foo/>` is treated like `<foo></foo>`. Body text is represented literally, without escapes.

Unlike official XML syntax, our format has some nice properties:

- YXML can be inlined directly into the string representation of text messages. It is orthogonal to UTF-8 encoding (since **X** and **Y** are part of 7-bit ASCII), orthogonal to Isabelle symbol notation (since it avoids `<>`), and independent of previous markup (marking text does not alter it, no escaping of `<>\&"'`).
- YXML parsing is straight-forward: first split the text into chunks separated by **X**, then split each chunk into sub-chunks separated by **Y**. Markup tags start with an empty sub-chunk, and a second empty sub-chunk indicates a close tag. Any other non-empty chunk consists of literal text.
  Thus efficient YXML parsers can be implemented in the programming language of choice in one afternoon, e.g. by taking the 1-page implementations in Isabelle/ML or Isabelle/Scala as blueprints.

Isabelle/ML already provides some library functions to inline YXML markup into text, using `Markup.markup: Markup.T -> string -> string` like this:

ML ⟪
```
  warning ("Potential problem:\n" ^ Markup.markup Markup.malformed
    ("bad variable variable " ^ Markup.markup Markup.hilite "x"))
```
⟫

JVM-based front-ends that receive Isabelle messages can use Isabelle/Scala library operations to parse the YXML representation, and work directly with XML trees. The above example results in the following structured message (in XML syntax), which also includes a header provided by the `warning` function:

```
<warning serial="409542" offset="1" end_offset="3" id="31">Potential problem:
<malformed>bad variable variable <hilite>x</hilite></malformed></warning>
```

Adding occasional markup to prover messages is adequate, but we do not even need this in most practical situations. Provers already provide standard library functions to produce string representations of formal entities (types, terms, theorems), such as our `Syntax.string_of_term` seen before. Since YXML is inlined into the ML string type, we can easily augment the standard operations to add markup transparently, without affecting user code that is well-behaved (refrains from analyzing formatted prover output).

Likewise, the ML function `Position.str_of: Position.T -> string` that prints text positions can be modified to include its own markup. Thus we get machine-readable positions in messages, and front-ends can easily display prover output with clickable spots or attach messages directly to the source view.

The following example illustrates such implicitly enhanced messages produced by existing text-oriented application code.

ML ⟪
```
  warning ("Term: " ^ Syntax.string_of_term ctxt t ^ Position.str_of pos)
```
⟫

Here the printed term `t` is $x + y$: it contains some undeclared variables that Isabelle highlights to warn the user. There is also some reference to the logical constant behind the "+" notation. The front-end receives the following output:

```
<warning serial="409564" offset="1" end_offset="3" id="37"Term:
<term><block indent="0"><block indent="0"><hilite><block
indent="0"><free><block indent="0">x</block></free></block></hilite>
<const name="Groups.plus_class.plus"><block
indent="0">+</block></const><break width="1"> </break><hilite><block
indent="0"><free><block
indent="0">y</block></free></block></hilite></block></block></term><position
offset="79" end_offset="82" id="-35"/></warning>
```

This means the prover now speaks XML natively, without changing the application code. Further markup content can be provided gradually, using XML as data language to represent certain aspects of the prover state, see also §3.

In retrospect, the re-use of text characters `<>\&"'` in official XML syntax (inherited from SGML) turns out as big obstacle for adoption in old-school applications. Earlier attempts of Isabelle/PGIP implementation [4] suffered from the difference of escaped vs. unescaped text in prover messages: the "polarity" of marked strings was often wrong, resulting in display errors or protocol crashes.

YXML can easily and efficiently repair the problems of concrete XML syntax. In a sense, our format returns to the original idea of "markup" before SGML and XML: information is attached to some text without altering it. One can also think of a physical text marker, which does not "escape" any text it touches.

3 XML Content

Stripped of its concrete syntax, what is the essence of XML and how can we use it to represent content of the prover adequately? At the bottom of the enormous XML standard documents, there is the bare tree structure consisting of text and markup elements (with name and attributes). This is specified in 3 lines of ML:

```
datatype tree =
  Elem of (string * (string * string) list) * tree list
| Text of string
```

The same is specified in 3 lines of Scala using *case classes* [7]:

```
sealed abstract class Tree
case class Elem(markup: (String, List[(String, String)], body: List[Tree])
  extends Tree
case class Text(content: String) extends Tree
```

This is naked, untyped XML — no more no less. Add-on standards for typed XML documents do exist (e.g. DTD, XML Schema, Relax NG), but there is no universal agreement, and availability in ML or Scala is limited. Nonetheless, the prover and its front-ends need to assign some meaning to XML trees.

Problem: Select suitable mechanisms to interpret raw XML documents and define concrete meaning for various prover document markup elements.

Solutions: From the many possibilities, we subsequently present schemes that are already used in Isabelle.

3.1 Typed Views on Untyped Data

We can think of XML trees as the raw material to represent content in memory, transport it between provers and front-ends, or store it in persistent databases and repositories. This is similar, to frugal s-expressions in LISP, although XML provides even less explicit data formats, basing everything on plain strings.

In ML and Scala we prefer to work with typed trees, following the tradition of algebraic datatypes. In practice there is a slight asymmetry: ML functions of the prover usually *produce* XML trees (by direct inlining of YXML, cf. §2.3), while the Scala/JVM front-ends *consume* them (parsing some tree content). Luckily each programming language provides sufficient means to represent these notions in simple manners, without requiring a full framework of "meta-programming" for externally specified XML datatypes.

ML *constructor functions* encode typed data as untyped markup, which can be inlined into YXML text.

Scala *extractor functions* analyse the content of untyped XML trees, after YXML parsing. This works via customized `unapply` methods of some Scala objects that model the notions of document content. (Symmetrically, it is also possible to define constructors via `apply` methods.)

For example, the ML function `Markup.proof_state: int -> Markup.T` encodes the number of pending sub-goals (of integer type) as textual markup node: `Markup.proof_state 42` is received by the JVM front-end as `<proof_state subgoals="42"/>` where the `tree` can be matched in Scala via `tree match { case Proof_State(i) => i }` to recover the integer 42.

To make such pattern matching work, Scala requires object `Proof_State` with some `unapply` method defined beforehand (e.g. in the prover library):

```
object Proof_State {
  def unapply(tree: XML.Tree): Option[Int] =
    tree match {
      case XML.Elem(Markup("proof_state",
        List(("subgoals", Markup.Int(i)))), Nil) => Some(i)
      case _ => None
    }
}
```

More ambitious ML/Scala connectivity could augment the Scala compiler by some plugin to produce these definitions from concise specifications. Such sophisticated meta-programming is outside our present scope, though.

Note that these data views are inherently partial: dynamic type mismatch can occur, and indicate some fault of the protocol infrastructure of the prover. User code usually works with statically-typed `Markup.proof_state` in ML and `Proof_State.unapply` in Scala, to minimize programming errors.

3.2 Augmented Oppen Pretty-Printing

Proof assistants mainly operate on symbolic term structures, which are eventually displayed to the user via *pretty printing*. This often works by some variant of Pretty-Printing according to Oppen [8]: text *atoms* and potential line *breaks* are arranged as nested *blocks* (with optional *indentation*). The resulting pretty tree can be formatted to match a given margin: the pretty printing algorithm inserts physical spaces and newlines as required.

The initial indication of breaks and blocks already constitutes a physical markup language, which can be augmented for our document-oriented architecture as follows.

1. Pretty blocks may also carry *logical markup*. The Isabelle/ML function `Pretty.markup: Markup.T -> Pretty.T list -> Pretty.T` inserts markup information into pretty trees, analogous to `Markup.markup` on plain text as seen before (§2.3).
2. In ML the pretty formatting is made *symbolic*. Instead of former physical layouting, the information about breaks and blocks is turned into XML markup elements `<break width="N"/>` and `<block indent="N">...</block>`, respectively. Together with the logical markup this constitutes an XML document, which is inlined into prover output by YXML encoding.
3. In Scala the physical formatting is now done directly on XML trees. Break and block markup elements are expanded to produce appropriate spaces and newlines within the body text; the remaining XML tree only contains the logical markup. The formatting algorithm imitates the former ML implementation closely, using typed views `Break(width)` and `Block(indent, body)` to recognize symbolic breaks and blocks in raw XML trees.
4. Final rendering works via XHTML/CSS, using the JVM-based Lobo/Cobra Web browser, for example. Markup elements `<foo>...</foo>` are turned into `<span class="foo">...</span>` (ignoring attributes) and then given to the browser together with an XHTML header and prover-specific style sheet. The XHTML text is essentially treated as preformatted, to retain the indentation and line breaking performed before.

Presently, Isabelle merely uses this XHTML rendering scheme to produce traditional highlighting of certain entities within printed terms, e.g. global variables in blue, local variables in brown, bound variables in green, undeclared variables with yellow background. Much more can be done by exploiting the full potential of XHTML/CSS, with boxes and table layouts, for example.

Since the Web browser allows to attach program code to HTML elements (either as JavaScript or JVM code, which can be produced in Scala), we could

interpret certain prover markup to fold/unfold sub-terms, or produce popups that reveal information about the formal entities at that point (the place of definition, documentation etc.). Using HTML input forms, one could also support simple interaction schemes based on prover output: the proof assistant presents some templates that the user can complete and insert into the source text.

4 Document Reconstruction

With the text markup and content infrastructure of §2 and §3 available, we can now look deeper into semantic prover information. Classic command-line provers only produce *output* in the form of messages, but in a document-oriented setting we can attach valuable information to the *input* via markup.

Conceptually, the sources define the true meaning of the text. By processing the sources, the prover explores the meaning incrementally, as represented by some internal configuration. Aspects of this semantic information can be attached to the original sources to help the user understand the text, or support tools that operate on the text systematically (e.g. via "refactoring").

This means, an approximation of the true semantic prover content is *reconstructed* as externally readable document. The prover as the only instance that can really analyze the formal text reveals important aspects about its content, without requiring front-ends to imitate substantial parts of the prover itself.

Inherent Complexity of Prover Syntax. Presenting sources of formal languages with rich semantic annotations is nothing new. For example, an IDE for Java understands the syntax and static semantics of the language (with scopes and type information for identifiers) in order to support systematic refactoring of the program text. Frameworks like Eclipse already provide powerful libraries (such as Xtext) to implement such functionality for user-defined languages.

Unfortunately, hardly anything like this works for common proof assistants. Apart from some lexical analysis and superficial parsing, there are many more syntax layers until fully typed λ-terms emerge internally. Some of these phases are well-defined, like Hindley-Milner type-inference over the order-sorted algebra of type-classes in Isabelle. Other phases are open-ended, providing computationally complete plugin-mechanisms: recent Isabelle allows to (re)define the type-discipline in user space, based on arbitrary ML code. The symmetric situation happens for output of λ-terms, until some printable text is eventually produced. Any of these mechanisms can depend on local declarations according to the scoping rules of the formal language.

The latter is illustrated by the following trivial example in Isabelle/Isar, with local mixfix declarations that modify some notation within proof blocks:

notepad
begin
  fix *foo* :: $'a \Rightarrow 'a \Rightarrow 'a$
  fix *bar* :: $'a \Rightarrow 'a \Rightarrow 'a$
  { write *foo* (**infix** $\cdot$ 70) **have** $x \cdot x = foo\ x\ x$ **.. }**
  { write *bar* (**infix** $\cdot$ 70) **have** $x \cdot x = bar\ x\ x$ **.. }**
end

Even more, Isabelle allows to *transform* arbitrary declarations by morphisms, to move between different formal contexts. We conclude that any prover IDE or theory browser that attempts to analyze the text directly is bound to fail!

Interestingly, the situation in Coq is similar. For example, there have been approaches to direct structural manipulation via "proof by pointing" [5], with strong assumptions about the content of Coq syntax trees. These were violated later as the prover evolved, causing the PCoq front-end [1] to break eventually.

Builders of standard IDE syntax plugins might shake their heads and demand that provers are restricted to decidable syntax, or deliver abstract syntax trees with full information. We argue that this is unrealistic: existing theory libraries with sophisticated notation would have to be reduced or discontinued. Moreover, current provers like Coq or Isabelle do not even address the full complexity of mathematical notation yet, and the trend is towards even more complex notational devices as can be seen in Matita [2], for example.

Problem: Find ways to transport semantic information through many sophisticated layers of prover syntax. Avoid strong assumptions about prover syntax trees. Observe the matter of fact that the prover cannot provide complete information, only some important aspects.

Solution: Careful inspection of the syntax hierarchy of sophisticated provers like Isabelle leads to the following observations:

1. Plain text is clearly observable by external tools, and the prover can agree with front-ends about precise addressing of text positions (cf. §2.2).
2. The main elements of the theory and proof language (called "commands" in Isabelle terminology) can be understood as *transactions*, which define clearly delimited updates on internal state. Assuming that execution of a transaction proceeds linearly (with monotonic increase of internal state information and without back-tracking) we can collect a *trace* of it as part of the result.
3. Trace information can be assembled into an external *markup tree* that is associated with the original piece of text in the transaction context.

Reporting of markup for certain positions generalizes the idea of printing text messages to the user. The Isabelle/ML message channels of `error`, `warning` etc. are extended by `Position.report: Position.T -> Markup.T -> unit` for maintaining the implicit markup tree of the current transaction. The prover merely needs to pass precise positions through to a point where certain formal content is discovered, and report an externalized version back to the sources.

Figure 1 illustrates why this makes an important conceptual difference: instead of demanding full information about sophisticated phases of parsing $f_1;f_2;f_3;f_4;f_5$ and printing $g_1;g_2;g_3;g_4;g_5$, we merely preserve position information up to the point of certain reports after $f_1;f_2$ and $f_1;f_2;f_3$, for example.

This means *less* information is passed through *some* of the input phases only. With this conceptual simplification in place, we now need to rework some central prover syntax modules, in order to provide useful reports. It might require extra

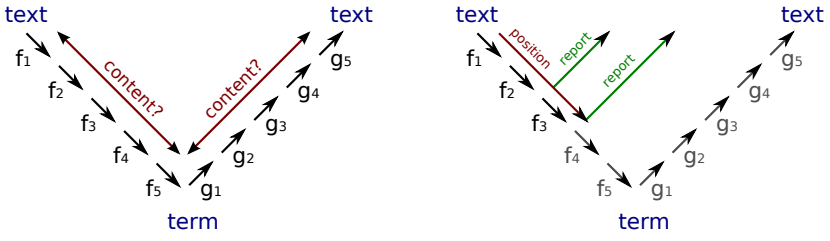


Fig. 1. Full content of input/output phases versus reports on some input phases

tricks to preserve precise position information in particular phases, e.g. the move from quoted "outer syntax" to "inner syntax" in Isabelle, where we revive the ancient ASCII DEL character (127) for padding.

In Isabelle the number of syntax phases is not 5, but approximately 12. For the inner syntax of terms and types this includes computationally complete mechanisms like so-called "translation functions", which are used for example to implement derived binders or implicit state-dependence as in Hoare logic notation. The regular type-checking phase has its own plugin-in concept, which typically includes extensions of Hindley-Milner type-inference, with "type improvement" and optional "coercion functions". It is unrealistic to expect that fully annotated syntax trees can be passed through all these phases, but position reports according to our scheme of figure 1 are quite feasible, after some minor reworking of the existing code base.

Document Markup for Prover IDEs. Isabelle2011 already reports about its "outer syntax" of tokens and command spans, including various binding positions that are directly accessible (e.g. the points where *foo* and *bar* were fixed in the previous example). The "inner syntax" of types and terms is more complex: tokens, raw parse trees, scopes of free versus bound variables are already accessible for reports, but type-information and sub-expression markup is still missing. The key idea for non-trivial term annotations, which is partially implemented in Isabelle repository versions after official Isabelle2011, is as follows: source positions within the raw term language are disguised as type constraints, and passed through the main syntax translation layers (where many user-defined transformations exist), until the point where Hindley-Milner type-inference happens. By construction of traditional lambda calculus with optional type-constraints, old syntax operations are expected to handle such extra annotations (but in practice it often means to repair a few omissions in user code).

The Poly/ML 5.4 compiler (by David Matthews) that is used with Isabelle2011 also produces significant reports about the results of static analysis of Standard ML. The following ML snippet inside Isabelle/Isar illustrates this, although the results are unavailable in the classic PDF version of this document.

ML ⟪
```
fun foo th = Thm.prems_of (th RS @{thm refl})
```
⟫

Processing these sources with the experimental Isabelle/jEdit Prover IDE of Isabelle2011 (see also figure 2), the document markup is presented as boxed sub-expressions, tooltips for inferred types, and hyper-links for referenced identifiers (such as the local `th` and the global `RS` operator from the Isabelle/ML library).

The syntax highlighting for tokens etc. is also based on precise semantic information from Isabelle: the editor merely retrieves it from the universal document markup tree.

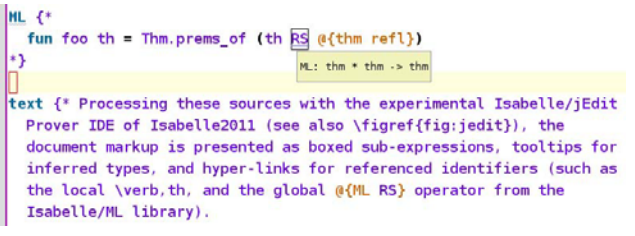


Fig. 2. IDE presentation of formal document content of these sources

Many more possibilities are conceivable, e.g. indicating variable scopes by highlighting binding and referencing positions simultaneously (as in common IDEs), or popup menus pointing to other formal documents that explain the referenced concepts (Isabelle manuals already provide suitable formal markup).

5 Conclusion and Related Work

We have shown how to augment old-school proof assistants to support text documents with rich semantic information, while minimizing the changes to the existing code base. The cultural background of provers implemented in ML (or Haskell) is preserved, we refrain from pervasive "XML-ization" of the programming style. Moreover, the increasingly popular Scala language on the JVM helps to extend the scope of the prover into a greater world.

This work can be seen as a continuation of earlier approaches to overcome the plain-text attitude of proof assistants: CtCoq/Pcoq [5], PLATΩ [11] and especially PGIP [4]. The efforts around PGIP and its (partial) implementation in Isabelle2004/2005 posed many problems that are now addressed quite differently than first anticipated: our problem statements and proposed solutions are best understood in the context of former struggles with PGIP. For example, PGIP demands to implement "proof-script parsing" on the spot without any further context. In contrast, our present approach lets the prover report information incrementally whenever it becomes available.

Matita [2] is probably the best-known proof assistant that was designed with some IDE support (based on Gtk) and advanced presentation formats (MathML) from the ground up. Thus it had the privilege of a fresh start, without a huge

existing code base to take care of. Nonetheless, the Matita authors report significant efforts to re-build basic infrastructure for rich mathematical output (such as their own MathML rendering engine in OCaml/GTk). Even after substantial efforts, the Matita front-end still resembles traditional Proof General [3]. In contrast, our general approach is to maximize re-use, by augmenting existing provers with minimal effort and simplifying connectivity to powerful front-ends (IDEs, browsers, Web frameworks), especially on the JVM. Moreover, our document model of annotated prover sources goes beyond the basic IDE functionality of Matita, which is more concerned about output of mathematical formulae than annotating input sources.

Earlier work on "XML-izing Mizar" [10] is based on the assumption that complete syntax trees in XML are feasible, in contrast to our observation of "inherent complexity of prover syntax" as discussed before. Despite its own builtin complexity, Mizar is a closed language and defining fixed XML formats for its tool chain turned reasonably easy. It also allows users to implement derivative tools, using XSLT for example. This more conventional way to externalize prover content would be hardly possible for the more complex and open-ended LCF family, notably Isabelle, Coq, and HOL.

Proviola [9] shares our goals to extract document content from prover runs. This is done for Coq, without modifying the prover itself (due to lack of access to its inner circle of developers). This limits the document content to traditional proof state and message output, for example. Our work can be understood as providing the missing link on the prover side: it could simplify projects like Proviola (and its greater perspective of prover Web clients and Wikis for formalized mathematics), and also provide more interesting content.

Ultimately, only the prover itself can reveal substantial document content, derived from its true semantic state. We have shown how this can be achieved in a profound manner, without rewriting major proof assistants from scratch. In the future we hope to see more proof assistants re-use these concepts, and more front-ends exploiting the content delivered by such document-oriented provers.

References

1. Amerkad, A., Bertot, Y., Pottier, L., Rideau, L.: Mathematics and proof presentation in Pcoq. In: Proceedings of Proof Transformation and Presentation and Proof Complexities, PTP 2001 (2001)
2. Asperti, A., Sacerdoti Coen, C., Tassi, E., Zacchiroli, S.: User interaction with the Matita proof assistant. Journal of Automated Reasoning 39(2) (2007)
3. Aspinall, D.: Proof General: A generic tool for proof development. In: Graf, S. (ed.) TACAS 2000. LNCS, vol. 1785, p. 38. Springer, Heidelberg (2000)
4. Aspinall, D., Lüth, C., Winterstein, D.: A framework for interactive proof. In: Kauers, M., Kerber, M., Miner, R., Windsteiger, W. (eds.) MKM/CALCULEMUS 2007. LNCS (LNAI), vol. 4573, pp. 161–175. Springer, Heidelberg (2007)
5. Bertot, Y., Théry, L.: A generic approach to building user interfaces for theorem provers. Journal of Symbolic Computation 25(7) (1998)

6. Gordon, M., Milner, R., Morris, L., Newey, M.C., Wadsworth, C.P.: A metalanguage for interactive proof in LCF. In: Principles of programming languages, POPL (1978)
7. Odersky, M., et al.: An overview of the Scala programming language. Technical Report IC/2004/64, EPF Lausanne (2004)
8. Oppen, D.C.: Pretty printing. ACM Transactions on Programming Languages and Systems 2(4) (1980)
9. Tankink, C., Geuvers, H., McKinna, J., Wiedijk, F.: Proviola: A tool for proof re-animation. In: Autexier, S., Calmet, J., Delahaye, D., Ion, P.D.F., Rideau, L., Rioboo, R., Sexton, A.P. (eds.) AISC 2010. LNCS, vol. 6167, pp. 440–454. Springer, Heidelberg (2010)
10. Urban, J.: XML-izing Mizar: Making semantic processing and presentation of MML eas. In: Kohlhase, M. (ed.) MKM 2005. LNCS (LNAI), vol. 3863, pp. 346–360. Springer, Heidelberg (2006)
11. Wagner, M., Autexier, S., Benzmüller, C.: PLATΩ: A mediator between text-editors and proof assistance systems. In: Autexier, S., Benzmüller, C. (eds.) User Interfaces for Theorem Provers (UITP 2006). ENTCS, vol. 174(2). Elsevier, Amsterdam (2007)
12. Wenzel, M.: Isabelle/Isar — a generic framework for human-readable proof documents. In: Matuszewski, R., Zalewska, A. (eds.) From Insight to Proof — Festschrift in Honour of Andrzej Trybulec. Studies in Logic, Grammar, and Rhetoric, vol. 10(23), University of Białystok (2007)
13. Wenzel, M.: Asynchronous proof processing with Isabelle/Scala and Isabelle/jEdit. In: Sacerdoti Coen, C., Aspinall, D. (eds.) User Interfaces for Theorem Provers (UITP 2010). ENTCS (July 2010), FLOC 2010 Satellite Workshop
14. Wenzel, M.: The Isabelle/Isar Implementation Manual (2011)
15. Wenzel, M., Chaieb, A.: SML with antiquotations embedded into Isabelle/Isar. In: Carette, J., Wiedijk, F. (eds.) Workshop on Programming Languages for Mechanized Mathematics (PLMMS 2007), Hagenberg, Austria (June 2007)
16. Wiedijk, F. (ed.): The Seventeen Provers of the World. LNCS (LNAI), vol. 3600. Springer, Heidelberg (2006)

Towards Formal Proof Script Refactoring

Iain Whiteside, David Aspinall, Lucas Dixon*, and Gudmund Grov

School of Informatics, University of Edinburgh,
Edinburgh EH8 9AB, Scotland

Abstract. We propose proof script refactorings as a robust tool for constructing, restructuring, and maintaining formal proof developments. We argue that a formal approach is vital, and illustrate by defining and proving correct a number of valuable refactorings in a simplified proof script and declarative proof language of our own design.

1 Introduction

Theorem proving in-the-small is popular for investigating small domain-specific logics and for teaching logic. With maturing technology, theorem proving in-the-large is becoming more common, with big formalisations built in both industry and academia. Notable examples in verification include a specification of the IP stack [15] and functional correctness of a microkernel [9]. Formalised mathematics is also feasible: Gonthier formalised the proof of the Four Colour Theorem (FCT) [7] and Hales has an ongoing project to formally prove the Kepler Conjecture [8]. The completed proof scripts (theorem prover inputs) range in length from around 10,000 lines to 200,000 lines, each proving hundreds or thousands of lemmas and representing several person-years of work.

It is encouraging that large developments are possible, but they are far from easy. Writing proofs is often harder than writing programs. Formal proofs are more complex, dense, and interdependent than similarly sized programs. Yet they are developed with primitive tools akin to those used for programming in the 1960s: often little more than basic text editors, with no high-level means of rapid construction, easy modification or browsing. Lack of modern support for proof construction and maintenance is a main reason that interactive provers are not used more widely; it makes them incredibly tedious to learn and use.

By contrast, maintenance of large programs is well supported by modern Software Engineering (SE) tools. Software systems of hundreds of thousands of lines of code can be managed with relative ease. One important SE technique is *refactoring*. A *refactoring* is a semantics-preserving transformation of code which improves design, structure or readability [6,12]. It may be pervasive, but routine. Ideally refactorings are tool-assisted: an algorithm checks safety pre-conditions before making global changes in one go. Many refactorings are simple operations with complex pre-conditions, e.g., *Rename* and *Move*. Our hypothesis is that by adapting and extending refactoring techniques from SE we can make

* Now at Google, New York.

J.H. Davenport et al. (Eds.): Calculemus/MKM 2011, LNAI 6824, pp. 260–275, 2011.

development and maintenance of formal proof scripts easier and more accessible to new users, as well as more productive for expert users. As anecdotal evidence to motivate our work, we note that Gonthier mentions, in [7], having to spend a number of months refactoring his FCT development by hand.

Ensuring correctness of proof refactorings is vital because proof scripts can take arbitrarily long to re- check, and unexpected changes to lemma statements can change the *meaning* of a development. It is also non-trivial; for example, complex tactics make analysis of dependencies difficult, as noted in [13], and notions of semantics for fully-blown proof scripts have not been well studied compared with programming language semantics. Even refactoring tools for programming languages, which have been studied for almost 20 years, are full of bugs [5].

In this work we study refactoring formally in a simple, generic proof script language of our own design in order to understand and overcome the main challenges. In particular, we use Hiproofs [4] as a generic notion of proof as it provides a clean theoretical base on which to build; furthermore, we believe the hierarchy offers opportunities for refactoring and proof understanding. We do not intend to cover all aspects of a practical implementation from the outset, but use it as an exploratory study into the viability, applicability, and challenges associated with refactoring.

Contributions. Our two main contributions are a generic proof script language, with a declarative proof language, and a formal treatment of a number of proof refactorings. Firstly, we give a formal semantics to the proof scripts and prove that declarative proofs construct valid proofs. In particular, we formalise a notion of *gaps* in a proof. Secondly, we define what we mean by proof script refactoring, and the appropriate notion of semantics preservation, and define several valuable refactorings, including *rename lemma* and *backward to forward*, which transforms a backwards-style proof into a forward-style one. Finally we prove that these refactorings are correct in a meaningful sense.

Outline of paper. In the next section, we introduce Hiproofs, as a representation for proof, and Hitac as an idealised tactic language on which we base our work. In Section 3 we introduce the proof script language, its semantics, and give a formal definition of proof script refactoring. We then, in Section 4, describe the declarative proof language and give an example of our proof scripts in Section 5. Section 6 describes a number of refactorings and we conclude in Section 7.

2 Background

Hiproofs are a hierarchical representation of the proof trees constructed by tactics. The hierarchy makes explicit the relationship between tactic calls and the proof tree constructed by these tactics. Hiproofs were first investigated by Denney et al in [4] and can be given a denotational semantics as a pair of forests, viewed as posets. One partial order provides a notion of hierarchy, the other of sequential composition. An abstract example of a Hiproof is given in Figure 1, where a, b, and c are called *atomic tactics* i.e. black boxes.

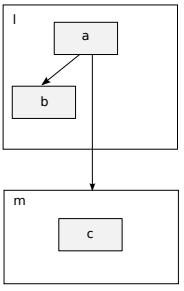


Fig. 1. A Hiproof

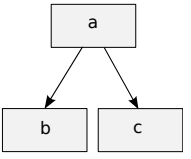


Fig. 2. The skeleton

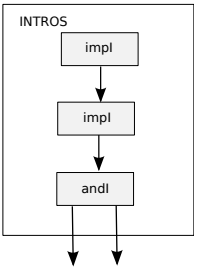


Fig. 3. INTROS

Figure 1 reads as follows: at the top, the abstract tactic l first applies an atomic tactic a. The tactic a produces two subgoals, the first of which is solved by the atomic tactic b within the application of l. Thus, the high-level view is that tactic l produces a single subgoal, which is then solved by the tactic m. The underlying proof tree, called the *skeleton*, is shown in Figure 2. Conditions placed upon construction of Hiproofs ensure that they can always be *unfolded* into the skeleton. More concretely, Figure 3 shows the application of an *INTROS* tactic as a Hiproof; the trailing edges are goals that must be solved by composing other Hiproofs. In [1], Aspinall et al gave an operational account of Hiproofs, based on *derivation systems*. They also introduced a tactic language, Hitac, which constructs Hiproofs. We describe this work now, as it provides the basis for what follows.

Derivation Systems. In this work we will not commit to a specific logical system; instead, we work within a *derivation system*, which can be thought of as a simple logical framework. It defines sets of *atomic goals* $\gamma \in \mathcal{G}$ and *atomic tactics* $a \in \mathcal{A}$. What we call an *atomic goal* is a judgement form in the underlying derivation system, and what we call an *atomic tactic* is an inference rule schema:

$$\frac{\gamma_1 \cdots \gamma_n}{\gamma} \; a \in \mathcal{A}$$

stating that the atomic tactic a, given subgoals $\gamma_1, \ldots, \gamma_n$, produces a proof of γ. We do not formalise how the rule schemata and instances are related.

Hiproofs. The concrete syntax of Hiproofs is defined by the grammar:

$$s \;::=\; a \mid id \mid [l]s \mid s \,;\, s \mid s \otimes s \mid \langle\rangle$$

Sequencing $(s \,;\, s)$ corresponds to composing boxes by arrows, tensor $(s \otimes s)$ places boxes side-by-side, and labelling $([l]s)$ introduces a new labelled box. Identity (id) and empty $(\langle\rangle)$ are units for ; and $\otimes$ respectively. Labelling binds weakest, then sequencing with tensor binding most tightly. We can now give a syntactic description of the Hiproof in Figure 1: $([l]\,a \,;\, b \otimes id) \,;\, [m]\; c$.

We say that a Hiproof is *valid* if it is well-formed and if atomic tactics are applied correctly. This notion is formalised with a validation relation $s \vdash g_1 \longrightarrow g_2$,

where g_i are lists of goals. This relation, defined below, states that the Hiproof s is a proof of g_1 with remaining subgoals g_2. Thus, Hiproofs can represent partial proofs. The γ_i are single goals and @ is list append.

$$\frac{\frac{\gamma_1 \ldots \gamma_n}{\gamma}\ a \in \mathcal{A}}{a \vdash [\gamma] \longrightarrow [\gamma_1, \ldots, \gamma_n]}\ \text{(V-Atomic)} \qquad \frac{s_1 \vdash g_1 \longrightarrow g \qquad s_2 \vdash g \longrightarrow g_2}{s_1\ ;\ s_2 \vdash g_1 \longrightarrow g_2}\ \text{(V-Seq)}$$

$$\frac{s \vdash [\gamma] \longrightarrow g}{[l]s \vdash [\gamma] \longrightarrow g}\ \text{(V-Label)} \qquad \frac{s_1 \vdash g_1 \longrightarrow g_1' \qquad s_2 \vdash g_2 \longrightarrow g_2'}{s_1 \otimes s_2 \vdash g_1 @ g_2 \longrightarrow g_1' @ g_2'}\ \text{(V-Tens)}$$

$$\frac{}{\langle\rangle \vdash [] \longrightarrow []}\ \text{(V-Empty)} \qquad \frac{}{id \vdash [\gamma] \longrightarrow [\gamma]}\ \text{(V-Id)}$$

Hitac. We extend the Hitac grammar from [1] with *lemma application*:

$$\begin{array}{rcl} t & ::= & a \mid id \mid [l]t \mid t\ ;\ t \mid t \otimes t \mid \langle\rangle \\ & \mid & assert\ \gamma \\ & \mid & t \mid t \\ & \mid & name(t, \ldots, t) \\ & \mid & X \\ & \mid & lem\ l \end{array}$$

In addition to the standard Hiproof constructs, goal assertions ($assert\ \gamma$) can control the flow; alternation ($t \mid t$) allows choice; and, defined tactics ($name(t, \ldots, t)$) and variables (X) allow us to build recursive tactic programs. Tactic evaluation is defined relative to a *proof environment*: $(\mathcal{T}, \mathcal{L})$. The lemma environment, $\mathcal{L} : name \to (goal \times s)$, is a map from lemma names to a goal and Hiproof pair; the tactic environment, $\mathcal{T} : name \to (\overline{X}, t)$, maps tactic names to their parameter list and Hitac tactic.

Evaluation of a tactic is defined as a relation $\langle g, t\rangle \Downarrow^t_{(\mathcal{T},\mathcal{L})} \langle s, g'\rangle$, which should be read as: 'the tactic t applied to the list of goals g returns a Hiproof s and remaining subgoals g', under $(\mathcal{T}, \mathcal{L})$'. We give the evaluation rules for *tensor*, *lemma application*, and *defined tactics* below, where we write $\overline{X}$ as shorthand for the list $[X_1, \ldots, X_n]$. For a full presentation of the semantics and a proof of Theorem 1, please refer to [1].

$$\frac{\langle g_1, t_1\rangle \Downarrow^t_{(\mathcal{T},\mathcal{L})} \langle s_1, g_1'\rangle \qquad \langle g_2, t_2\rangle \Downarrow^t_{(\mathcal{T},\mathcal{L})} \langle s_2, g_2'\rangle}{\langle g_1 @ g_2, t_1 \otimes t_2\rangle \Downarrow^t_{(\mathcal{T},\mathcal{L})} \langle s_1 \otimes s_2, g_1' @ g_2'\rangle}$$

$$\frac{\mathcal{T}(name) = (\overline{X}, t) \qquad \langle g, t[t_1/X_1, \ldots, t_n/X_n]\rangle \Downarrow^t_{(\mathcal{T},\mathcal{L})} \langle s, g'\rangle}{\langle g, name(t_1, \ldots, t_n)\rangle \Downarrow^t_{(\mathcal{T},\mathcal{L})} \langle s, g'\rangle}$$

$$\frac{\mathcal{L}(l) = (\gamma, s) \qquad s \vdash \gamma \longrightarrow []}{\langle \gamma, lem\ l\rangle \Downarrow^t_{(\mathcal{T},\mathcal{L})} \langle [lem\ l]s, []\rangle}$$

Theorem 1 (Correctness of big-step semantics). *If* $\langle g_1, t\rangle \Downarrow^t_{(\mathcal{T},\mathcal{L})} \langle s, g_2\rangle$ *then* $s \vdash g_1 \longrightarrow g_2$.

3 Proof Scripts

Proof scripts allow us to build and extend proof environments. For simplicity, proof scripts consist of only tactic definitions and lemmas although real systems need other elements (e.g., definitions, syntax, comments). The grammar for proof scripts is:

$$
\begin{array}{lcl@{\qquad}lcl}
\mathit{proofscript} & ::= & \textbf{script}\ \mathit{name} & \mathit{scriptobj} & ::= & \textbf{begin} \\
 & & \quad \mathit{scriptobj}^* & & | & \textbf{tacdef}\ \mathit{name}(X_1, \ldots, X_n) := t \\
 & & \textbf{end} & & | & \textbf{lemma}\ \mathit{name}\textbf{:}\ \mathit{goal} \\
 & & & & & \quad \mathit{prf}
\end{array}
$$

Proof scripts essentially consist of a sequence of lemmas and tactics, within **begin** and **end** tags. Lemmas are named and consist of a goal and a formal proof *prf*; parameterised tactics can be defined, where the X_i are the tactic variables occurring within t and must be instantiated when the tactic is used.

Top level evaluation of proof scripts is defined by:

$$\frac{\vdash \mathit{scriptobjs} \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle}{\vdash \textbf{script}\ \mathit{name}\ \mathit{scriptobjs}\ \textbf{end} \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle}\ \text{(PS-Script)}$$

where the judgement $\vdash \mathit{scriptobjs} \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle$, which operates on a sequence of script objects, is defined in Figure 4. The evaluation relation states that the script *proofscript* can be evaluated resulting in an *environment* $\langle \mathcal{T}, \mathcal{L} \rangle$. When we do not need to explicitly refer to the environment, we will write $\vdash \mathit{proofscript}$ to mean $\exists\ \mathcal{T}\ \mathcal{L}.\ \vdash \mathit{proofscript} \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle$. We say a proof script is *well-formed* iff $\vdash \mathit{proofscript}$. Well-formedness does not imply that the proofs are complete: there may be gaps; *proof checking* is a low-level process that would ensure the Hiproofs constructed have no trailing edges. For brevity, we often use P to refer to proof scripts.

The rule PS-Begin ensures scripts start with a **begin** and initialises an empty environment. Tactics (or lemmas) extend $\mathcal{T}$ (or $\mathcal{L}$) as long as they satisfy the preconditions. The functions *names* and *variables* are defined recursively on scripts and tactics and the relation $\mathit{scriptobjs} \vdash t$ checks for well-formedness of

$$\vdash \textbf{begin} \Downarrow \langle \{\}, \{\} \rangle \qquad \text{(PS-Begin)}$$

$$\frac{\vdash \mathit{scriptobjs} \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle \qquad n \notin \mathit{names}(\mathit{scriptobjs}) \qquad \langle \gamma, \mathit{prf} \rangle \Downarrow^{p}_{(\mathcal{T},\mathcal{L})} \langle s \rangle}{\vdash \mathit{scriptobjs}\ \textbf{lemma}\ n\textbf{:}\ \gamma\ \mathit{prf} \Downarrow \langle \mathcal{T}, \mathcal{L}[n \mapsto (\gamma, [n]s)] \rangle}\ \text{(PS-Lem)}$$

$$\frac{\vdash \mathit{scriptobjs} \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle \qquad n \notin \mathit{names}(\mathit{scriptobjs}) \qquad \mathit{variables}(t) \subseteq \overline{X} \qquad \mathit{scriptobjs} \vdash t}{\vdash \mathit{scriptobjs}\ \textbf{tacdef}\ n(\overline{X}) := t \Downarrow \langle \mathcal{T}[n \mapsto (\overline{X}, [n]t)], \mathcal{L} \rangle}\ \text{(PS-Tac)}$$

Fig. 4. Proof script evaluation

tactics: ensuring that only tactics and lemmas defined above it in the script can be used. The notation $\mathcal{T}[n \mapsto (\overline{X}, [n]t)]$ means extending the map, $\mathcal{T}$, by adding an element. For a lemma, the important precondition is that $\langle \gamma, prf \rangle \Downarrow^{p}_{(\mathcal{T},\mathcal{L})} \langle s \rangle$. This is the *proof relation*, which states that when you apply *prf* to the goal γ it results in a Hiproof, s. We will instantiate *prf* in the next section, but it could be any proof language associated with a *valid* proof relation i.e. one that constructs valid Hiproofs. Both lemmas and tactics are labelled with their name when added to the environment allowing us to see, in the hierarchy, where tactics and lemmas have been applied.

3.1 Proof Script Refactoring

We first define a more general notion of proof script transformation as follows:

Definition 1 (Proof Script Transformation). *A pair $(\mathcal{P}, \mathcal{O})$, where $\mathcal{P}$ is a predicate, called the precondition, and $\mathcal{O} : proofscript \rightarrow proofscript$, is a **proof script transformation** if, for all scripts P, such that $\vdash P$ and $\mathcal{P}(P)$, we have $\vdash \mathcal{O}(P)$.*

That is, applying a proof script transformation, as long as the precondition $\mathcal{P}$ holds, guarantees that it preserves well-formedness.

We take the view that the statements proved within lemmas are the key semantics objects in a script, motivating the following definition:

Definition 2 (Statement Preservation). *A proof script transformation, $(\mathcal{P}, \mathcal{O})$, is **statement preserving** if for all scripts P such that $\mathcal{P}(P)$ and $\vdash P \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle$ then $\vdash \mathcal{O}(P) \Downarrow \langle \mathcal{T}', \mathcal{L}' \rangle$ and we have:*

$$\forall\ l \text{ if } \ \mathcal{L}(l) = (\gamma, s) \ \text{ then } \ \exists\ l'\ s'\ s.t.\ \mathcal{L}'(l') = (\gamma, s').$$

That is, we prove at least the same statements after a transformation as before. Thus, we have:

Definition 3 (Proof Script Refactoring). *A **proof script refactoring** is a proof script transformation that is statement preserving.*

4 A Declarative Proof Language

To explore the refactoring possibilities within proofs, we describe a declarative proof language and give it a semantics which constructs valid Hiproofs. We describe *prf* with the following grammar:

prf	::=	**proof(** *rule* **)**	*stmt*	::=	$\langle\rangle$
		stmt*		|	[*name*]**:{** *prf* **}**
		qed		|	**apply** *rule*
	|	**gap**		|	**show** *name* : *goal prf*
				|	**have** *name* : *goal prf*
rule	::=	*t*		|	**from** *name** **show** *goal* **by** *rule*

The core component of the language is a *proof block*: **proof(***rule***)** *stmts* **qed**. Proof blocks operate on a single goal, applying the initial rule before solving the resulting subgoals by the statements inside it. The key statement is **show**, which solves the goal it is applied to. The *empty* statement $\langle\rangle$ operates on an empty list, finishing off a proof; tactics can be applied directly with **apply**; forward proof is possible by using **have** to extend the environment then the **from**...**show**...**by** construct to perform the forward step. Hierarchy can be added, using the labelling construct: [*name*]**:{** *prf* **}**; finally, goals can be skipped with the **gap** command. Syntactic conveniences, for example, **by** *rule* $\equiv$ **proof(***rule***)** $\langle\rangle$ **qed** can be introduced. In Figure 5, we give a big-step semantics to declarative proofs with the relation $\langle g, prf\rangle \Downarrow^{p}_{(\mathcal{T},\mathcal{L})} \langle s\rangle$. A proof *prf* applied to a list of goals g will result in a Hiproof s. Top level *prf* evaluations always operate on a single goal, and the evaluation rules in Figure 5 reflect this. Statement lists are evaluated one at a time; the statement being operated on directly is highlighted. We use :: to refer to the *cons* list constructor.

Proof blocks operate on singleton goal lists and are evaluated by executing the initial tactic, then feeding the resulting subgoals into the enclosed statements. The **gap** proof construct also operates on singleton goals, placing an *identity*

$$\frac{\langle [\gamma], t\rangle \Downarrow^{t}_{(\mathcal{T},\mathcal{L})} \langle s_1, g\rangle \qquad \langle g, stmts\rangle \Downarrow^{p}_{(\mathcal{T},\mathcal{L})} \langle s_2\rangle}{\langle [\gamma], \textbf{proof(}t\textbf{)}\ stmts\ \textbf{qed}\rangle \Downarrow^{p}_{(\mathcal{T},\mathcal{L})} \langle s_1\ ;\ s_2\rangle} \quad \text{(B-Prf-Block)}$$

$$\langle [\gamma], \textbf{gap}\rangle \Downarrow^{p}_{(\mathcal{T},\mathcal{L})} \langle id\rangle \quad \text{(B-Prf-Gap)}$$

$$\langle [], \langle\rangle\rangle \Downarrow^{p}_{(\mathcal{T},\mathcal{L})} \langle\langle\rangle\rangle \quad \text{(B-Prf-Empty)}$$

$$\frac{\langle [\gamma], prf\rangle \Downarrow^{p}_{(\mathcal{T},\mathcal{L})} \langle s_1\rangle \qquad \langle g, stmts\rangle \Downarrow^{p}_{(\mathcal{T},\mathcal{L})} \langle s_2\rangle}{\langle \gamma :: g, [\mathrm{l}]\textbf{:\{}\ prf\ \textbf{\}}\ stmts\rangle \Downarrow^{p}_{(\mathcal{T},\mathcal{L})} \langle([l]s_1)\ \otimes\ s_2\rangle} \quad \text{(B-Prf-Lab)}$$

$$\frac{\langle g_1, t\rangle \Downarrow^{t}_{(\mathcal{T},\mathcal{L})} \langle s_1, g_2\rangle \qquad \langle g_2, stmts\rangle \Downarrow^{p}_{(\mathcal{T},\mathcal{L})} \langle s_2\rangle}{\langle g_1, \textbf{apply}\ t\ stmts\rangle \Downarrow^{p}_{(\mathcal{T},\mathcal{L})} \langle s_1\ ;\ s_2\rangle} \quad \text{(B-Prf-App)}$$

$$\frac{\langle [\gamma], prf\rangle \Downarrow^{p}_{(\mathcal{T},\mathcal{L})} \langle s_1\rangle \qquad \langle g, stmts\rangle \Downarrow^{p}_{(\mathcal{T},\mathcal{L})} \langle s_2\rangle}{\langle \gamma :: g, \textbf{show}\ name\colon \gamma\ prf\ \ stmts\rangle \Downarrow^{p}_{(\mathcal{T},\mathcal{L})} \langle s_1\ \otimes\ s_2\rangle} \quad \text{(B-Prf-Show)}$$

$$\frac{\langle [\gamma], prf\rangle \Downarrow^{p}_{(\mathcal{T},\mathcal{L})} \langle s_1\rangle \qquad name \notin names(\mathcal{T}) \cup names(\mathcal{L}) \qquad \langle g, stmts\rangle \Downarrow^{p}_{(\mathcal{T},\mathcal{L}[name \longmapsto (\gamma, s_1)])} \langle s\rangle}{\langle g, \textbf{have}\ name\colon \gamma\ prf\ \ stmts\rangle \Downarrow^{p}_{(\mathcal{T},\mathcal{L})} \langle s\rangle} \quad \text{(B-Prf-Have)}$$

$$\frac{\begin{array}{c}\mathcal{L}(n_1) = (\gamma_1, s_1)\ \ \ldots\ \ \mathcal{L}(n_n) = (\gamma_n, s_n) \\ \langle [\gamma], t\rangle \Downarrow^{t}_{(\mathcal{T},\mathcal{L})} \langle s, [\gamma_1, \ldots, \gamma_n]\rangle \qquad \langle g, stmts\rangle \Downarrow^{p}_{(\mathcal{T},\mathcal{L})} \langle s'\rangle\end{array}}{\begin{array}{c}\langle \gamma :: g, \textbf{from}\ n_1\ \ldots\ n_n\ \textbf{show}\ name\colon \gamma\ \textbf{by}\ t\ \ stmts\rangle \Downarrow_{(\mathcal{T},\mathcal{L})} \\ \langle (s\ ;(s_1 \otimes \ldots \otimes s_n)) \otimes s', []\rangle\end{array}} \quad \text{(B-Prf-From)}$$

Fig. 5. Declarative proof language big-step semantics

Hiproof to feed the goal out. In B-Prf-Have, we see that the intermediate step is added to the proof environment. We enforce the new name to be unique, but this is not necessary: if we drop this restriction we can have local overloading in the current proof block. This does, however force us to provide more complex preconditions and transformation rules for the refactorings.

The rule B-Prf-From is the most complicated. The first set of preconditions check that the names used exist in the lemma environment; the next ensures that the tactic justification returns exactly the stated goals that the names refer to; the third simply evaluates the remaining statements. Finally the Hiproof is constructed by tensoring together all of the Hiproofs for each individual subgoal and placing them after the Hiproof resulting from the tactic application. In order to ensure that *valid* Hiproofs are constructed, we prove:

Theorem 2 (Soundness of big-step semantics). *If* $\langle \gamma, prf \rangle \Downarrow^{p}_{(\mathcal{T},\mathcal{L})} \langle s \rangle$ *then* $s \vdash \gamma \longrightarrow g$ *for some* g. *Moreover, if* prf *is* ***gap-free*** *then* $s \vdash \gamma \longrightarrow []$.

Proof: We proceed by induction on the height of the derivations. For empty, the rule B-Prf- Empty matches V-Empty directly with $g = []$. Gaps are validated with V-Id and $g = [\gamma]$. For B-PRf-Block, Theorem 1 and the induction hypothesis allow us to apply V-Seq. Similarly, with B-Prf-Show we use the induction hypothesis twice and V-Tens, with g being the concatenation of both. The other cases are similar. When the proof is gap-free, we note that B-Prf-Gap is the only base case to introduce a discrepancy between Hiproof validation and tactic evaluation, thus g must be []. In fact, g is exactly the 'gapped' goals.

Theorem 3 (Completeness of big-step semantics). *If* $s \vdash \gamma \longrightarrow []$ *for a given environment* $(\mathcal{T}, \mathcal{L})$ *then there exists a gap- free* prf *such that* $\langle \gamma, prf \rangle \Downarrow^{p} \langle s \rangle (\mathcal{T}, \mathcal{L})$.

Proof: If s is a Hiproof such that $s \vdash \gamma \longrightarrow []$ then, trivially, '**by** s' works when we consider the Hiproof as a Hitac tactic.

5 Example

In order to make these languages more concrete, we provide a small example proof script. Space restrictions require any such example is necessarily trivial but we hope it conveys some of the main features. We instantiate a derivation system with first order logic, with atomic tactics given by the well-known natural deduction rules, a few of which are given below:

$$\frac{\Gamma, P \vdash Q}{\Gamma \vdash P \to Q}\ impI \qquad \frac{\Gamma \vdash P \qquad \Gamma, Q \vdash R}{\Gamma, P \to Q \vdash R}\ impE \qquad \frac{}{\Gamma, P \vdash P}\ ax \qquad \frac{\Gamma, P, Q \vdash R}{\Gamma, P \wedge Q \vdash R}\ conjE$$

$$\frac{\Gamma \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash P \wedge Q}\ conjI \qquad \frac{\Gamma \vdash P[x := x_0]}{\Gamma \vdash \forall\ x.P}\ allI \qquad \frac{\Gamma, P[x := t] \vdash Q}{\Gamma, \forall\ x.P \vdash Q}\ allE$$

An example script, defining two tactics and proving two lemmas is given in Figure 6. Note that we elide the empty statement in both lemmas.

```
script example begin
  tacdef REPEAT(X) := X ; (REPEAT(X) | id)
  tacdef intros := REPEAT(impI | allI | conjI)

  lemma lemma1 : ⊢ P ∧ Q → Q ∧ P
  proof(intros)
   show q : {P ∧ Q} ⊢ Q by (conjE ; ax)
   show p : {P ∧ Q} ⊢ P by (conjE ; ax)
  qed

  lemma lemma2 : ⊢ (∀ x. P x → Q x) → (∀ x. P x) → (∀ x. Q x)
  proof(intros)
   show {∀ x. P x → Q x, ∀ x. P x} ⊢ Q x
   proof(REPEAT(allE))
    show {P x → Q x, P x} ⊢ Q x by impE ; (ax ⊗ ax)
   qed
  qed
end
```

Fig. 6. Example proof script

6 Refactorings

We are now ready to refactor proof scripts. We describe *rename lemma*, *swap objects*, *transform proof*, *backward proof to forward proof*, and *extract subproof* in some detail. We summarise, in Figure 7, the main refactorings we have considered thus far. Finally in Section 6.6 we show how Figure 6 could be refactored.

6.1 Rename Lemma

Renaming a lemma may seem like a trivial action but, if that lemma has been applied multiple times in a proof development, the new name must be propagated and must not clash with any other names in the development. This makes it a

Generalise Tactic	Replace a *subtactic* with a var, creating a more general tactic.
Fold/Unfold Proof	Declarative proofs can be collapsed into raw tactic applications and vice-versa.
Fold/Unfold Tactic	Fold or unfold a defined tactic.
Fill Gap	Replace a **gap** with a proof that solves the goal.
Add/Rem Hierarchy	Introduce or remove labelled boxes to a proof.
Safe Delete	Delete a lemma or tactic as long as it is not used.
Copy	Copy a lemma or tactic.
Rename	Rename a lemma or tactic.
Backward to Forward	Convert a backward proof into a forward proof.
Have to Lemma	Lift a **have** statement up to the status of lemma.
Extract Subproof	Extract a subproof of a goal into a lemma.

Fig. 7. Summary of refactorings

tedious and error-prone task for humans. The refactoring takes three parameters: a script to operate on, an old lemma name, o, and a new lemma name, n.

Preconditions. In common with all other refactorings that we are currently considering, *rename lemma* has the precondition that the proof script P that it acts on must be well-formed i.e. $\vdash P$. We must also ensure that there are no name clashes with the new name: $n \notin names(P)$.

Transformation rules. We define this transformation on the structure of proof scripts. There are three classes of rules operating on: a *proofscript*, a *prf*, and a t. We give some cases of the action on *proofscript* and t in Figures 8 and 9. The *rename tactic* refactoring is analagous, except we must take into account possibly recursive tactics.

Correctness. We want to prove that this operation is indeed a refactoring:

Theorem 4 (Rename Lemma Correctness). *If, for a proof script P and old name o and new name n that satisfy the preconditions above and*

$$P \xrightarrow{rl(o,n)} P' \quad \textit{and} \quad \vdash P \Downarrow \langle \mathcal{T}, \mathcal{L} \rangle \quad \textit{then} \quad \vdash P' \Downarrow \langle \mathcal{T}', \mathcal{L}' \rangle$$

and we have: $\forall\ l$ if $\mathcal{L}(l) = (\gamma, s)$ then $\exists\ l'\ s'$ *s.t.* $\mathcal{L}'(l') = (\gamma, s')$.

$$\frac{scriptobjs \xrightarrow{rl(o,n)} scriptobjs'}{\textbf{script}\ name\ scriptobjs\ \textbf{end} \xrightarrow{rl(o,n)} \textbf{script}\ name\ scriptobjs'\ \textbf{end}}$$

$$\textbf{begin} \xrightarrow{rl(o,n)} \textbf{begin}$$

$$scriptobjs\ \textbf{lemma}\ o\textbf{:}\ \gamma\ prf \xrightarrow{rl(o,n)} scriptobjs\ \textbf{lemma}\ n\textbf{:}\ \gamma\ prf$$

$$\frac{scriptobjs \xrightarrow{rl(o,n)} scriptobjs' \qquad prf \xrightarrow{rl(o,n)} prf' \qquad o \neq ln}{scriptobjs\ \textbf{lemma}\ ln\textbf{:}\ \gamma\ prf \xrightarrow{rl(o,n)} scriptobjs'\ \textbf{lemma}\ ln\textbf{:}\ \gamma\ prf'}$$

$$\frac{scriptobjs \xrightarrow{rl(o,n)} scriptobjs' \qquad t \xrightarrow{rl(o,n)} t'}{scriptobjs\ \textbf{tacdef}\ tn(\overline{X}) := t \xrightarrow{rl(o,n)} scriptobjs\ \textbf{tacdef}\ tn(\overline{X}) := t'}$$

Fig. 8. Script level transformations

Proof. We prove that $\vdash P' \Downarrow \langle \mathcal{T}', \mathcal{L}' \rangle$ by induction on the transformation rules. For each rule, we show that if the script evaluates before the rule is applied, then it evaluates to an equivalent environment afterwards; this motivates our need for the precondition as it is required as a premiss for one of the evaluation rules. We can then see that $\mathcal{L}'$ satisfies the statement preservation property.

$$\frac{t_1 \xrightarrow{rl(o,n)} t_1' \qquad t_2 \xrightarrow{rl(o,n)} t_2'}{t_1 \ ; \ t_2 \xrightarrow{rl(o,n)} t_1' \ ; \ t_2'}$$

$$\frac{l \neq o}{lem\ l \xrightarrow{rl(o,n)} lem\ l}$$

$$lem\ o \xrightarrow{rl(o,n)} lem\ n$$

Fig. 9. Tactic level transformations

6.2 Swap Objects

We can swap two adjacent objects if they have no dependency. In Figure 6 we can swap the two lemmas, but not the definitions of *intros* and *REPEAT*. We can repeat this refactoring to get the more general *move object* refactoring. To simplify presentation, we assume that we are swapping two adjacent lemmas (although the general refactoring covers all four cases). *Swap object* takes two parameters: the name of a lemma, x and the script P. We take the convention that the named lemma is to be moved up one place.

Preconditions. Given $posn(x,\ P) = n$, $objAt(n-1,\ P) = y$, and $envAt(n,\ P) = (\mathcal{T}, \mathcal{L})$. If $proof(x) = prf$, $\mathcal{L}(x) = (\gamma, s)$, and $\langle \gamma, prf \rangle \Downarrow^p_{(\mathcal{T},\mathcal{L})} \langle s \rangle$ then we must have $\langle \gamma, prf \rangle \Downarrow^p_{(\mathcal{T}, del(y,\ \mathcal{L}))} \langle s \rangle$.

That is, the lemma x can still be proved in an environment without y. This formulation of the preconditions simplifies the correctness proof, but it could also be described purely syntactically for our language; however, for languages with sophisticated tactics like *auto*, we would need to use the semantic information. All the functions used here are easily defined on scripts.

Transformation. We only show two of the transformation rules:

$$\frac{obj_2\ \ objs \xrightarrow{swap(x)} objs' \qquad nameOf(obj_2) \neq x}{obj_1\ \ obj_2\ \ objs \xrightarrow{swap(x)} obj_1\ \ objs'}$$

$$obj_1\ \ \textbf{lemma}\ \text{x}\text{:}\ \gamma\ prf\ \ objs \xrightarrow{swap(x)} \textbf{lemma}\ \text{x}\text{:}\ \gamma\ prf\ \ obj_1\ \ objs$$

Correctness. We elide the proof that this is indeed a refactoring.

6.3 Transform Proof

This refactoring is an *enabling refactoring* for the ones to follow. Essentially, it takes a *proof transformation*, $\mathcal{R} : prf \rightarrow prf$, a lemma name, n, as parameters and applies $\mathcal{R}$ to the proof of that lemma, leaving everything else untouched. The precondition for this refactoring is that the proof transformation preserves provability. We do not give the transformation rules here as they are straightforward. This is clearly a refactoring: the provability precondition matches directly with the premiss for PS-Lem.

6.4 Backward Proof to Forward Proof

This refactoring transforms a proof *prf* that is in the form of Figure 10 to the form of Figure 11.

proof(t **)**
  show *goal1* : γ_1 prf_1
  $\vdots$
  show *goaln* : γ_n prf_n
qed

Fig. 10. Before

proof
  have *goal1* : γ_1 prf_1
  $\vdots$
  have *goaln* : γ_n prf_n
  from *goal1* ... *goaln* **show** γ **by** t
qed

Fig. 11. After

Preconditions. We have one precondition: there are no '**apply** t' steps within the proof block. Rather than a technical limitation, it simplifies the presentation of the rules below. In order to remove it, we would have to convert any **apply** steps into a declarative proof format, which is another refactoring.

Transformation rules. We describe the refactoring using a set of transformation rules, a subset of which is given below:

$$\frac{stmts \xrightarrow{b2f} stmts' \qquad [n_1, \ldots, n_n] = shows(stmts)}{\textbf{proof(}t\textbf{)}\ stmts\ \textbf{qed} \xrightarrow{b2f} \textbf{proof}\ stmts'\ \textbf{from}\ n_1\ \ldots\ n_n\ \textbf{show}\ \gamma\ \textbf{by}\ t\ \textbf{qed}}$$

$$\frac{stmts \xrightarrow{b2f} stmts'}{\textbf{show}\ name\textbf{:}\ \gamma\ prf\ \ stmts \xrightarrow{b2f} \textbf{have}\ name\textbf{:}\ \gamma\ prf\ \ stmts'}$$

This time, the transformation rules work only on a *prf*. In order to apply the refactoring at the script level, we use *transform proof*. The function *shows* constructs a list of all the goals.

Correctness. Since this refactoring applies only to the proof and the precondition for *transform proof* is that *backward to forward* preserves provability, we only need to show:

Theorem 5 (Provability Preservation of Backward Proof to Forward Proof). *If prf is a declarative proof of* γ *satisfying the preconditions of backward to forward then*

$$prf \xrightarrow{b2f} prf' \quad \textit{and} \quad \langle \gamma, prf' \rangle \Downarrow^{p}_{(\mathcal{T},\mathcal{L})} \langle s' \rangle$$

Proof. Since **show** statements get transformed to **have** statements by the refactoring, and by the precondition that the names are fresh in the environment, we can guarantee that all the names are in the environment when the **from**...**show**... **by** is executed. This succeeds because the names map directly to the subgoals produced by t initially.

6.5 Extract Subproof

In this more complex refactoring, we show how a proof of a subgoal within a lemma can be extracted as a lemma in its own right. It is, in fact, a composition of two simpler refactorings:

Show to Have: transforms a **show** statement into a **have** statement and replaces the proof of the show statement with a '**by** *lem n*', where n is the user supplied name for the have statement.

Have to Lemma: moves a have statement up to the top level of the script: there are no preconditions and no change is required to the rest of the proof. This refactoring would be useful if an intermediate lemma is more widely applicable.

Figures 12 to 14 show how this refactoring proceeds. We do not give a more formal description here. It is worth noting that we can compose these refactorings because *Have to Lemma* does not have any preconditions; however, in general, to ensure that two refactorings can be composed to form another correct refactoring we must be able to prove that the preconditions for the second refactoring are always satisfied. In future work, we intend to investigate composition using *postconditions* of refactorings.

6.6 Example Refactoring

Finally, using *rename lemma*, *fold tactic*, *backward proof to forward proof*, and *swap lemma* we can transform the proof script in Figure 6 into Figure 15. In particular, we rename *lemma1* to *conj_comm* and *lemma2* to *all_mp*, which better reflect their meaning, and swap their position. We have used *backward proof*

lemma *lname* : γ
proof
⋮
show *gi* : γ_i
prf_i
⋮
qed

Fig. 12. Before

lemma *lname* : γ
proof
⋮
have n : γ_i prf_i
show *gi* : γ_i **by** *lem n*
⋮
qed

Fig. 13. Step one

lemma n : γ_i
prf_i

lemma *lname* : γ
proof
⋮
show *gi* : γ_i **by** *lem n*
⋮
qed

Fig. 14. After

```
script example begin
  tacdef REPEAT(X) := X ; (REPEAT(X) | id)
  tacdef intros := REPEAT(impI | allI | conjI)
  tacdef conjEax := conjE ; ax

  lemma all_mp : ⊢ (∀ x. P x → Q x) → (∀ x. P x) → (∀ x. Q x)
  proof(intros)
   show {∀ x. P x → Q x, ∀ x. P x} ⊢ Q x
   proof(REPEAT(allE))
    show {P x → Q x, P x} ⊢ Q x by impE ; (ax ⊗ ax)
   qed
  qed

  lemma conj_comm : ⊢ P ∧ Q → Q ∧ P
  proof
   have q : {P ∧ Q} ⊢ Q by conjEax
   have p : {P ∧ Q} ⊢ P by conjEax
   from q p show {P ∧ Q} ⊢ Q ∧ P by intros
  qed
end
```

Fig. 15. Refactored example proof script

to forward proof to transform *conj_comm* and also *fold tactic* to replace the identical applications of *conjE ; ax* with a named tactic called *conjEax*.

7 Related Work and Conclusions

This paper introduces proof script refactoring as a way to make structured changes to a proof development. We describe a number of valuable refactorings including *rename lemma*, and *extract subproof* for a simple proof script and declarative proof language. We believe that the formal approach we take is necessary: the time required for proof-checking and the risk of changing the meaning of a lemma makes the correctness of refactorings vital. While we believe our work on proof script refactoring is unique, there is a large literature in the domain of programming language refactoring. Fowler takes a test-based approach to refactoring in [6]; this book, widely considered to be the 'handbook of refactoring', consists of over 70 refactorings with a detailed description of the motivation for each refactoring and how to carry it out safely. We note that refactoring has benefited from formal study in prototypical languages: Cornélio et al specify refactorings for a subset of Java, called ROOL [3] and prove semantics preservation using a set of basic algebraic laws, expressing equivalences between objects. In [10], Li and Thompson discuss a formal specification of Haskell refactorings based on an abstract representation of a program and provide a proof that the semantics of the program are preserved during the refactoring.

Mens and Tourwe give a much more detailed survey of programming language refactoring in [11]. There is also an interest in refactoring formal specifications. For example, in [16], the authors suggest refactorings for Z specifications based on experience on several large- scale projects. The effects of refactorings in Z are closely related to those in a formal proof script as, when schemas are refactored, this has an effect on all proofs relying of properties of these definitions. Schairer and Hutter describe a transformation framework for formal specifications in [14]. Although they do not consider semantics preservation, their approach is similar to our own, working independent of any logical system.

Declarative proof languages were pioneered by the Mizar system [17]. Our declarative language is designed to incorporate many of the core features of popular derivative languages, such as Isar for Isabelle [18], and C-zar for Coq [2]. However, due to our abstract approach, we do not have declarative statements that refer to the logical structure of a goal. For example, we only have **show** instead of the Isar-style **fix**...**assume**...**show** or the direct mapping between inference rules and C-zar statements such as **assume**. Our semantic approach to gaps is more closely related to Isar, where lemmas can be proved with gaps; C-zar, by comparison, does not allow a final 'qed' with unproven subgoals.

Further work. There are a number of extensions to our language we wish to consider. We would like to investigate a simple module system and define refactorings that operate at the module level. We could, for example, merge modules or move lemmas from one module to another. A more sophisticated *logical framework* would enable us to refactor statements directly, allowing us, for example, to *remove assumption*, if it is unused. Our proof script language also needs to be extended: we have yet to deal with *definitions* and *axioms*, both of which come with refactorings. On the practical side, we would like to create an implementation of a refactoring tool for our prototype language instantiated with a real derivation system. From the refactorings we have looked at so far, we have noticed that many can be built from smaller, *atomic* refactorings. The *move object* refactoring can be built from repetitions of the simpler *swap object* refactoring. We would like to investigate this further, perhaps coming up with a refactoring calculus. Fowler, in [6], discusses *bad smells* in code that indicate that a refactoring would be desirable. Typical examples are *duplicated code* and *long method*. These translate nicely into *duplicated proof steps* and *long proofs*. We are interested in discovering more, proof-specific smells. One particular methodology would be to analyse the version history of large development under Subversion or CVS control.

Acknowledgements. The authors would like to thank the anonymous reviewers for their helpful suggestions. The first author was supported by Microsoft Research through its PhD Scholarship Programme. All authors are grateful for the support of EPSRC Platform grant EPE/005713/1. The fourth author was also supported by EPSRC grant EP/H024204/1.

References

1. Aspinall, D., Denney, E., Lüth, C.: Tactics for hierarchical proof. Mathematics in Computer Science 3, 309–330 (2010)
2. Corbineau, P.: A declarative language for the Coq proof assistant. In: Miculan, M., Scagnetto, I., Honsell, F. (eds.) TYPES 2007. LNCS, vol. 4941, pp. 69–84. Springer, Heidelberg (2008)
3. Cornlio, M., Cavalcanti, A., Sampaio, A.: Refactoring by transformation. Electronic Notes in Theoretical Computer Science 70(3), 311–330 (2002)
4. Denney, E., Power, J., Tourlas, K.: Hiproofs: A hierarchical notion of proof tree. Electr. Notes Theor. Comput. Sci. 155, 341–359 (2006)
5. Ettinger, R., Verbaere, M.: Refactoring bugs in Eclipse, IntelliJ IDEA and Visual Studio (2005), http://progtools.comlab.ox.ac.uk/projects/refactoring/bugreports
6. Fowler, M.: Refactoring: improving the design of existing code. Addison-Wesley, Reading (1999)
7. Gonthier, G.: The Four Colour Theorem: Engineering of a formal proof. In: Kapur, D. (ed.) ASCM 2007. LNCS (LNAI), vol. 5081, pp. 333–333. Springer, Heidelberg (2008)
8. Hales, T.C.: Formal proof. Notices of the AMS 55, 1370–1380 (2008)
9. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. In: Proceedings of the 22nd ACM Symposium on OSP, pp. 207–220. ACM, New York (October 2009)
10. Li, H., Thompson, S.: Formalisation of Haskell Refactorings. In: Trends in Functional Programming (September 2005)
11. Mens, T., Tourwe, T.: A survey of software refactoring. IEEE Trans. Softw. Eng. 30(2), 126–139 (2004)
12. Opdyke, W.F.: Refactoring object-oriented frameworks. PhD thesis, University of Illinois, Champaign, IL, USA (1992)
13. Pons, O., Bertot, Y., Rideau, L.: Notions of dependency in proof assistants. In: User Interfaces for Theorem Provers, UITP (1998)
14. Schairer, A., Hutter, D.: Proof transformations for evolutionary formal software development. In: Kirchner, H., Ringeissen, C. (eds.) AMAST 2002. LNCS, vol. 2422, pp. 441–456. Springer, Heidelberg (2002)
15. Serjantov, A., Sewell, P., Wansbrough, K.: The UDP calculus: Rigorous semantics for real networking. In: Kobayashi, N., Babu, C. S. (eds.) TACS 2001. LNCS, vol. 2215, pp. 535–559. Springer, Heidelberg (2001)
16. Stepney, S., Polack, F., Toyn, I.: Refactoring in maintenance and development of Z specifications. Electr. Notes Theor. Comput. Sci. 70(3) (2002)
17. Josef, U., Grzegorz, B.: Presenting and explaining Mizar. Electron. Notes Theor. Comput. Sci. 174(2), 63–74 (2007)
18. Wenzel, M.: Isar - a generic interpretative approach to readable formal proof documents. In: Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., Théry, L. (eds.) TPHOLs 1999. LNCS, vol. 1690, pp. 167–184. Springer, Heidelberg (1999)

mizar-items: Exploring Fine-Grained Dependencies in the Mizar Mathematical Library

Jesse Alama*

Center for Artificial Intelligence,
New University of Lisbon
j.alama@fct.unl.pt

The MML is one of the largest collection of formalized mathematical knowledge that has been developed with various interactive proof assistants. It comprises more than 1100 "articles" summing to nearly 2.5 million lines of text, each consisting of a unified collection of mathematical definitions and proofs. Semantically, it contains more than 50000 theorems and more than 10000 definitions expressed using more than 7000 symbols. It thus offers a fascinating corpus on which one could carry out a number of experiments. This note discusses a system for computing fine-grained dependencies among the contents of the MML. For an overview of Mizar, see [3]; for a discussion of some successful initial experiments carried out with the help of mizar-items, see [1,2].

We say that a definition, or a theorem, ϕ *depends* on some definition, lemma or other theorem ψ, (or equivalently, that ψ is a *dependency* of ϕ) if ϕ "needs" ψ to exist or hold. The main way such a "need" arises is that the well-formedness or the justification of provability does not hold in the absence of ψ. Other senses of mathematical "dependency" are available that are related to what we describe here, but which mizar-items does not treat (at present). One might be interested, for example, in the space of all *logically possible* proofs of a certain result. Our interest is, to put it philosophically, intensional rather than extensional: we are interested in computing what minimally accounts for the success of a *specific* mathematical proof that has been formalized in the Mizar language. The extensional problem is what we are after, in the long run, but since we must work with specific formalizations of mathematical knowledge, we need to take an intensional approach.

The primary motivation behind mizar-items was the lack of a tool for giving a complete answer to the question of what, precisely, a Mizar text depends upon. This turns out to be rather non-trivial task. The difficulty stems primarily from various mechanisms (such as type inference) for making Mizar texts "smoother" for the author and human consumer because these mechanisms, by suppressing inferences—sometimes trivial, other times mathematically significant—can be "exposed" only through much computation.

* The author was supported by the FCT project "Dialogical Foundations of Semantics" (DiFoS) in the ESF EuroCoRes programme LogICCC (FCT LogICCC/0001/2007). The author thanks Artur Kornłowicz, Karol Pąk and Josef Urban for offering their Mizar expertise.

J.H. Davenport et al. (Eds.): Calculemus/MKM 2011, LNAI 6824, pp. 276–277, 2011.

Naturally, not all items in the vast Mizar library are equally interesting. mizar-items and its accompanying website (see below) was motivated by the problem of discovering dependency information not for arbitrary Mizar items, but specifically for those with substantial mathematical or foundational value, such as the Jordan curve theorem, the axiom of choice, the existence of strongly inaccessible cardinal numbers, or Euler's polyhedron formula (to name only a biased handful of examples). The fine-grained dependency data exposed by mizar-items could also be used in theory exploration and reverse mathematics [5] or Lakatos-style [4] investigations of necessary and sufficient conditions for mathematical theorems.

We compute the fine-grained dependency graph for the MML by starting with an over-approximation of what is known to be sufficient for an item to be Mizar-verifiable and then successively refining this over-approximation toward a minimal set of sufficient conditions. The method can be fairly characterized as brute-force: for each Mizar item, we successively hide implicit information normally kept hidden from a human Mizar formalizer, then see whether Mizar can still verify it. It turns out that this approach is rather slow; we needed to develop various heuristics to make the brute-force computation smarter.

mizar-items is accompanied by a website,

http://mizar.cs.ualberta.ca/mizar-items/

for exploring these dependencies. With the site one can view any particular Mizar item and see precisely what it depends upon (and what depends on the item). With the dependency graph, one can ask such queries as: *Is there a path between two given items? Do all paths from one item to another pass through a given intermediate node? Are there any paths between two given items that do not pass through a given node?*

To facilitate exploration, one can start by visiting a list of selected interesting entry points into the vast Mizar library.

mizar-items is a collection of programs in Common Lisp, Perl, Pascal, as well as shell scripts. The code is available online at

https://github.com/jessealama/mizar-items

(The Pascal sources are not included here: They are part of the Mizar code base, which, at present, is not publicly available.)

References

1. Alama, J., Kuehlwein, D., Tsivtsivadze, E., Urban, J., Heskes, T.: Premise selection for mathematics by corpus analysis and kernel methods (preprint, submitted)
2. Alama, J., Mamane, L., Urban, J.: Dependencies in formal mathematics (preprint, submitted)
3. Grabowski, A., Kornilowicz, A., Naumowicz, A.: Mizar in a nutshell. Journal of Formalized Reasoning 3(2), 153–245 (2010), http://jfr.cib.unibo.it/article/view/1980/1356
4. Lakatos, I.: Proofs and Refutations. Cambridge University Press, Cambridge (1976)
5. Simpson, S.G.: Subsystems of Second Order Arithmetic, 2nd edn. Perspectives in Mathematical Logic. Springer, Heidelberg (2009)

Formalization of Formal Topology by Means of the Interactive Theorem Prover Matita

Andrea Asperti[1], Maria Emilia Maietti[2], Claudio Sacerdoti Coen[1], Giovanni Sambin[2], and Silvio Valentini[2]

[1] Department of Computer Science, University of Bologna
{asperti,sacerdot}@cs.unibo.it

[2] Department of Mathematics, University of Padova
{maietti,sambin,silvio}@math.unipd.it

The project entitled "Formalization of Formal Topology by means of the interactive theorem prover Matita" is an official bilateral project between the Universities of Padova and Bologna, funded by the former, active from March 2008 until August 2010. The project aimed to bring together and exploit the synergic collaboration of two communities of researchers, both centered around constructive type theory: on one side the Logic Group at the University of Padova, focused on developing formal, pointfree topology within a constructive and predicative framework; on the other side, the Helm group at the University of Bologna, developing the Matita Interactive Theorem Prover [2], a young proof assistant based on the Calculus of Inductive Constructions as its logical foundation. The idea of the project was to formalize and check the new approach to formal topology being developed in Padova by means of Matita, with the aim on one side to assess the truly foundational nature of the theoretical framework (i.e. its reduction to notions so elementary to be easily understood by an automatic device), and on the other to drive the development of Matita, testing the tool on a non trivial set of mathematical results, and addressing from an orginal theoretical perspective some key problems of constructive interactive proving (general recursion, extensionality, quotients, ...).

The project is a rare example of a significant collaboration between mathematicians and computer scientists in handling mathematical knowledge.

It is worth to emphasize that the interest in the formalization from the mathematical perspective *is not* in the automatic verification of the results (on which mathematicians are already largely confident with) but in the phenomenological goal to investigate the most natural way to organize a new foundational framework in a coherent set of interconnected components, their mutual relations and dependencies, our interaction with these representations, and their influence on the concrete mathematical experience (see [1]). Formalization is neither a goal nor a technique, but first and foremost a methodology.

The formalization work was mainly focused on Overlap Algebras [4,3], new algebraic structures designed to ease reasoning about subsets within intuitionistic logic. The main result checked in Matita [5] is the embedding of a suitable category of topological spaces into a category of generalized point-free topologies,

J.H. Davenport et al. (Eds.): Calculemus/MKM 2011, LNAI 6824, pp. 278–280, 2011.

which is an improvement on the usual adjunction between topological spaces and locales. The formalization of this result drove several major improvements of Matita, discussed in [5].

A major feature of type theory (and proof checking systems based on this framework) is that functions are live entities, in the sense that they can be effectively computed. However, in presence of dependent types and for consistency reasons, one has to restrict to a subset of *total* computable funtions (typically based on some well founded recursion principle), that prevents from programming in a truly natural functional style. For this reason, the encoding of general recursion and its (simulated) behaviour is a major topic for interactive proving. In [6], a new approach to this problem based on the use of inductively generated formal topologies is proposed. The work is based on previous results in [9], relating the notion of covering with that of well-founded part of a relation.

Another challenging problem in type theory/interactive proving is extensionality, and especially an extensional treatment of set theory with quotients. In [7], a two-level theory to formalize constructive mathematics is presented, developing ideas already outlined in [8]. One level is given by an intensional type theory, called Minimal Type Theory. This theory extends a previous version with collections. The other level is given by an extensional set theory that is interpreted in the first one by means of a quotient model. This two-level theory has two main features: it is minimal among the most relevant foundations for constructive mathematics; it is constructive thanks to the way the extensional level is linked to the intensional one which fulfills the "proofs as programs" paradigm and acts as a programming language. The possibility to integrate this two level approach in Matita is still under investigation.

The code of the formalization developed in the project can be found on-line at the address `http://matita.cs.unibo.it/library.shtml`, under the section "The Basic Picture".

References

1. Asperti, A., Avigad, J.: Zen and the art of formalization. Mathematical Structures in Computer Science 21 (to appear, 2011)
2. Asperti, A., Coen, C.S., Tassi, E., Zacchiroli, S.: User interaction with the Matita proof assistant. Journal of Automated Reasoning 39(2), 109–139 (2007)
3. Ciraulo, F., Maietti, M.E., Toto, P.: Constructive version of Boolean Algebra. IGPL (to appear, 2011)
4. Ciraulo, F., Sambin, G.: The overlap algebra of regular opens. Journal of Pure and Applied Algebra 214, 1988–1995 (2010)
5. Coen, C.S., Tassi, E.: Formalizing Overlap Algebras in Matita. Mathematical Structures in Computer Science 21, 1–31 (2011)
6. Coen, C.S., Valentini, S.: General recursion and formal topology. In: Proceedings Workshop on Partiality and Recursion in Interactive Theorem Provers. EPTCS, vol. 43, pp. 65–75 (2010)

7. Maietti, M.E.: A minimalist two-level foundation for constructive mathematics. Annals of Pure and Applied Logic 160(3), 319–354 (2009)
8. Maietti, M.E., Sambin, G.: From Sets and Types to Topology and Analysis. In: Toward A Minimalist Foundation for Constructive Mathematics, ch. 6. Oxford University Press, Oxford (2005)
9. Valentini, S.: Cantor theorem and friends, in logical form. Annals of Pure and Applied Logic 163 (to appear, 2011)

Project EuDML — A First Year Demonstration

José Borbinha[1], Thierry Bouche[2], Aleksander Nowiński[3], and Petr Sojka[4]

[1] INESC-ID, Portugal
jlb@ist.utl.pt
[2] Institut Fourier (UMR 5582) & Cellule Mathdoc (UMS 5638), Université Joseph-Fourier, (Grenoble 1), B.P. 74, 38402 Saint-Martin d'Hères, France
thierry.bouche@ujf-grenoble.fr
[3] Interdisciplinary Center for Mathematical and Computational Modelling, University of Warsaw, ul. Pawińskiego 5A, 02-106 Warsaw, Poland
A.Nowinski@icm.edu.pl
[4] Masaryk University, Faculty of Informatics, Botanická 68a, 602 00 Brno, Czech Republic
sojka@fi.muni.cz

Abstract. This demonstration describes the results of the first year of the EuDML project, an initiative building a new multilingual service for searching and browsing the content of existing European portals of mathematical content. We demonstrate the first versions and proofs of concept of the EuDML portal, its contents' aggregator, and a toolset for added value.

About EuDML. — EuDML, the European Digital Mathematics Library (WWW.EUDML.EU), is a project that will build a new multilingual service for searching and browsing the content of existing European mathematical portals [5,1]. It will be based on a rich metadata repository, aggregating metadata and full text of heterogeneous and multilingual collections of digitised and born digital content (articles, books, theses, etc.). The service will merge and augment the information about each document from each collection, and also will match documents and references across the entire combined library. Entities such as authors, bibliographic references and mathematical concepts will be singled out and linked to matching items in the collections; similar mechanisms will be provided as public web-services so that end-users or other external services will be able to discover and link to EuDML items. This way, EuDML will be a new major international player in the emerging landscape of scientific information discovery services, enabled for reuse in new added value chains. EuDML is partially funded by the Competitiveness and Innovation Framework Programme of the European Commission (CIP ICT PSP Digital Libraries), grant agreement no. 250.503.

The EuDML Service Architecture. — The EuDML system can be summarised by the use cases represented in Figure 1. The ultimate purpose will be to serve End Users, who can search and browse anonymously, or can register for personalised services. A set of Business Workers are intended to maintain the services and content, while external business partners contribute their content (bibliographic data and full texts for indexing and added value services).

J.H. Davenport et al. (Eds.): Calculemus/MKM 2011, LNAI 6824, pp. 281–284, 2011.

The EuDML Portal. — The first version of the EuDML portal can be accessed from the EuDML website.[1] So far, there are no access restrictions, as all the services are available for anonymous users. This demonstration contains approximately 55,000 documents, provided by a group of partners (CEDRAM, DML-CZ, DML-E, ELibM, GDZ, NUMDAM, Portugaliæ Mathematica, and RusDML).

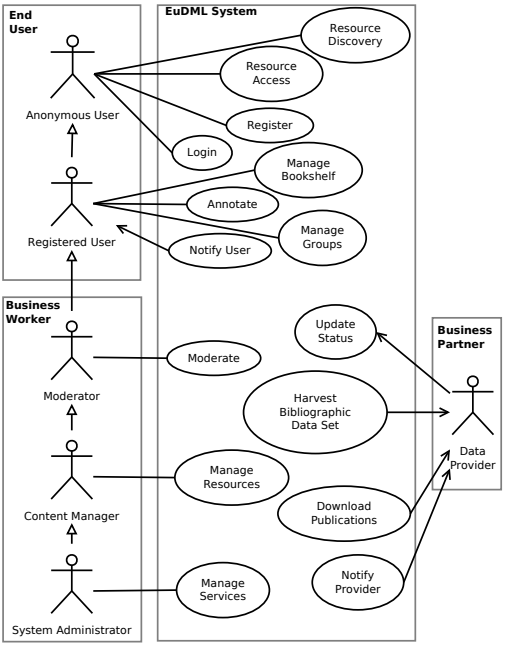


Fig. 1. The EuDML Use Cases

EuDML Content Aggregation. — One of the first project's result was a detailed analysis of the existing content formats and metadata schemas used throughout partnering projects and content providers. Informed by this study, a specification for a EuDML schema, heavily based on NLM JATS[2], was written down.

REPOX is a framework to manage data sets. It comprises multiple channels to import data from providers, services to transform data between schemas according to specified rules, and services to expose the results to the exterior. REPOX allows to monitor OAI-PMH[3] servers and schedule data ingests.

Instances of REPOX for EuDML are currently running at Instituto Superior Técnico (Lisbon) and Cellule MathDoc (Grenoble). These instances aggregate the bibliographic metadata collected up to now, converted to EuDML format, and exploited partially within the portal.

REPOX is complemented by the EuDML Profile Report, a service to inspect and create statistics and metrics on data quality, including whether the data conforms to particular standards or patterns. All these can be accessed from the page set-up in the EuDML website.[4]

The EuDML Enhancer and Association Toolsets. — This demonstration also comprises tools gathered or produced by EuDML partners as building bricks of enhancer tools, whose functionality should check, normalize and enhance metadata collected from partners, including Zentralblatt MATH, or extracted from the analysis of the full text of

[1] Go to HTTP://WWW.EUDML.EU/FIRST-YEAR-DEMOS#SYSTEM

[2] Journal Archiving and Interchange Tag Suite from the US National Library of Medicine, cf. HTTP://DTD.NLM.NIH.GOV/

[3] The Open Archives Initiative Protocol for Metadata Harvesting, cf. HTTP://WWW.OPENARCHIVES.ORG/

[4] Go to HTTP://WWW.EUDML.EU/FIRST-YEAR-DEMOS#AGGREGATION

items in the EuDML collections. Demonstration web pages allow testing and evaluation of prototypes of thirteen tools.[5]

This toolset consists of solutions for OCR, information extraction, content analysis, data conversion and document refinement. At this stage, more tools are being developed and tested mostly at the technology providers' sites, with well defined interfaces allowing further integration into subsystems of the EuDML core system site. As a next step, these tools will be merged into bigger components and installed in the central EuDML system, together with recently developed search of mathematical formulae [4].

Another set of tools targets tasks as interlinking scientifically related items in EuDML: turning citations into links [2], computing semantically similar papers or plagiarism candidate paper pairs [3]. Similar tools for linking to items in external services such as reviewing databases will be developed.

The EuDML User Interface Design and Tools. — The success of the project depends not only on the amount of aggregated data, but on the user experience as well. During the interface design process, a usability study has been performed. The study identified typical usage patterns, so it was possible to design an interface oriented toward scholar's productivity. The initial version of the portal covers the basic functionality of the digital library: searching and browsing collections, downloading the content, etc. Next step of the project is to add Web 2.0 functionalities, allowing to annotate and comment documents, and to share them with others, using both internal mechanisms and external services. The user interface design is also focused on providing efficient support for the mathematical content, both in presentation and user input (search or annotations).

The user interface is developed in Java, and is based on Spring framework[6]. Both EuDML repository services and web interface are based on the Yadda toolkit developed in ICM[7], customised and extended for required functionality.

Final Note. — The services described here are expected to evolve rapidly during the remaining project's lifetime. Up-to-date services and documentation will be linked from the resources page on our web site.[8]

References

1. Bouche, T.: Introducing EuDML—The European Digital Mathematics Library. EMS Newsletter 76, 11–16 (2010)
2. Goutorbe, C.: Document Interlinking in a Digital Math Library. In: Sojka, P. (ed.) Proceedings of DML 2009, Grand Bend, Ontario, CA, pp. 85–94. Masaryk University (July 2009), http://dml.cz/dmlcz/702560
3. Řehůřek, R., Sojka, P.: Software Framework for Topic Modelling with Large Corpora. In: Proceedings of LREC 2010 workshop New Challenges for NLP Frameworks, Valletta, Malta, pp. 45–50. ELRA (May 2010), http://is.muni.cz/publication/884893/en, http://nlp.fi.muni.cz/projekty/gensim

[5] See http://www.eudml.eu/first-year-demos#toolset
[6] See http://www.springsource.org/
[7] See http://yaddainfo.icm.edu.pl/
[8] See http://www.eudml.eu/resources

4. Sojka, P., Líška, M.: Indexing and Searching Mathematics in Digital Libraries (May 2011), accepted for CICM 2011 track MKM in this LNAI issue
5. Sylwestrzak, W., Borbinha, J., Bouche, T., Nowiński, A., Sojka, P.: EuDML—Towards the European Digital Mathematics Library. In: Sojka, P. (ed.) Proceedings of DML 2010, Paris, France, pp. 11–24. Masaryk University (July 2010), http://dml.cz/dmlcz/702569

A Symbolic Companion for Interactive Geometric Systems*

Francisco Botana

Departamento de Matemática Aplicada I, Universidad de Vigo,
Campus A Xunqueira, 36005 Pontevedra, Spain
fbotana@uvigo.es

System Description

We consider the problem of enriching a dynamic geometry system with new features from the field of Automated Deduction in Geometry. A prototype based on Sage, `sagemath.org`, that extends the current capabilities of the interactive environment GeoGebra, `geogebra.org`, is presented. The prototype provides a deeper knowledge of the different geometric objects in a construction. More precisely, an algebraic symbolic approach based on Groebner Bases is followed to implement a substitute for the numerical approach for property checking and locus plotting used by GeoGebra. As a result the system provides a *certified* answer in the case of a geometric query or the algebraic equation of the locus in the case of a locus construction. Note that knowing the equation of a locus is necessary to derive new geometric objects from the locus.

The prototype works only in connection with GeoGebra in its current form, but it can be easily adapted to any system allowing open access to a textual representation of its constructions. This includes in particular the case of systems exporting their constructions in *Intergeo* File Format, standard supported by most European dynamic geometry software.

The system automates two common tasks in the dynamic geometry paradigm: proving and locus discovery. GeoGebra was chosen as test tool for our parametric approach [1] due to its increasing relevance in the world of dynamic geometry, making it a *de facto* standard in the field. Roughly, given a GeoGebra construction with a locus or a boolean command checking some mathematical relation between elements, the system reads its XML description, and translates it to a parametric setting. The key point of this parametric representation of the construction is that the description of dependent elements is not given by their current positions, but by their algebraic relations. The system algebraically process this parametric representations to return the locus equation or a statement about the truth of the query. See [2] for the algorithm followed to obtain the locus equation and [3] for the one followed to test the truth of a statement.

Regarding the scope of admissible GeoGebra constructions, the system can currently process constructions involving the following elements: Free Point, Midpoint, Point, Segment, Line, OrthogonalLine, Circle, Intersect, Locus and Relation between Two Objects (parallelism, perpendicularity).

* Partially supported by the Spanish MICINN, grant MTM2008-04699-C03-03/MTM.

J.H. Davenport et al. (Eds.): Calculemus/MKM 2011, LNAI 6824, pp. 285–286, 2011.

Using the System

The system can be used in two different (web–based) ways. We consider first the method recommended for testing the tool. One has to login into a Sage server (`http://{alpha.}sagenb.org` or `https://193.146.36.205:9011`) as `cicm11`, password `test`. Once logged in, one just has to open one of the provided worksheets and follow the instructions there. The second way of using the tool involves what is known as *simple* Sage server which is included in the system to demonstrate the feasibility of allowing remote computations in an interactive geometric system. Since our system uses the friendly environment offered by Sage and also makes an extensive use of *Singular* functions, it is not currently possible to import the system as a Python library (although Sage developers plan to incorporate it in the near future). In that case, the simple server would be available as a temporal remote access solution for GeoGebra users.

Finally, at `http://webs.uvigo.es/fbotana/CICM11/`, all necessary files to test and use the system, including the worksheet files for standalone use and a 5' tutorial video, are available.

Examples

To facilitate the tool testing, two GeoGebra constructions are provided in the URL above as examples for each task. In the case of Pascal's limaçon, the system finds the appropriate quartic curve plus a circle as an extra degenerated part of the locus. For the second locus, a conchoid, it provides both branches of the locus. It must be noted that the followed elimination based approach does not return the exact locus, but its Zariski closure. So, the result equation $f = 0$ must be understood as in the sentence "the locus is, or is contained, in the affine variety $f = 0$".

The examples involving the proof task show identical results to those provided by GeoGebra. However, due to the algebraic techniques used, they are mathematically correct while the GeoGebra statements are based on numeric computations (as explicitly stated in GeoGebra 4.0) and hence are prone to inaccuracies.

References

1. Botana, F.: On the Parametric Representation of Dynamic Geometry Constructions. In: Murgante, B., Gervasi, O., Iglesias, A., Taniar, D., Apduhan, B.O. (eds.) ICCSA 2011, Part IV. LNCS, vol. 6785, pp. 342–352. Springer, Heidelberg (2011), `http://webs.uvigo.es/fbotana/s11.pdf`
2. Botana, F., Valcarce, J.L.: A Software Tool for the Investigation of Plane Loci. Math. Comput. Simul. 61(2), 139–152 (2003)
3. Recio, T., Vélez, M.P.: Automatic Discovery of Theorems in Elementary Geometry. J. Autom. Reasoning 23, 63–82 (1999)

MathScheme: Project Description*

Jacques Carette, William M. Farmer, and Russell O'Connor

Department of Computing and Software
McMaster University
Hamilton, Ontario, Canada
{carette,wmfarmer}@mcmaster.ca, roconnor@theorem.ca

The mission of mechanized mathematics is to develop software systems that support the process people use to create, explore, connect, and apply mathematics. Working mathematicians routinely leverage a powerful synergy between deduction and computation. The artificial division between (axiomatic) theorem proving systems and (algorithmic) computer algebra systems has broken this synergy. To significantly advance mechanized mathematics, this synergy needs to be recaptured within a single framework. MathScheme [6] is a long-term project being pursued at McMaster University with the aim of producing such a framework in which formal deduction and symbolic computation are tightly integrated. In the short-term, we are developing tools and techniques to support this approach, with the long-term objective to produce a new system.

Towards this aim, we have already developed several techniques, with some laying the theoretical foundations of our framework, while others are implementation techniques. In particular, we rely on biform theories and an expressive logic (Chiron) for grounding. We rely on various meta-programming techniques as well as the increased safety offered by a modern statically typed programming language (Objective Caml [8]) to greatly simplify our implementation burden.

A **biform theory** [1,3] is a combination of an axiomatic theory and an algorithmic theory. It is the basic unit of mathematical knowledge that consists of a set of *concepts*, *transformers*, and *facts*. The concepts are symbols that denote mathematical values and, together with the transformers, form a language L for the theory. The transformers are programs whose input and output are expressions of L. Transformers represent syntax-manipulating operations such as inference and computation rules. The facts are statements expressed in L about the concepts and transformers. In a typical biform theory, the concepts are classified as primitive or defined, the transformers as primitive or derived, and the facts as axioms, definitions, or theorems. A pure axiomatic theory is a biform theory with no transformers, and a pure algorithmic theory is a biform theory with no facts or only facts about the transformers.

Since transformers manipulate the syntax of expressions, biform theories are difficult to formalize in a traditional logic without the means to reason about syntax. **Chiron** [4,5] is a derivative of von-Neumann-Bernays-Gödel (NBG) set theory that is intended to be a practical, general-purpose logic for mechanizing mathematics. It is equipped with a type system that includes dependent types,

* This research was supported by NSERC.

J.H. Davenport et al. (Eds.): Calculemus/MKM 2011, LNAI 6824, pp. 287–288, 2011.

subtypes, and possibly empty types. It handles undefined expressions according to the *traditional approach to undefinedness*. Its most noteworthy component is a facility for reasoning about the syntactic structure of expressions using quotation and evaluation *à la* Lisp. We have an implementation [7] of Chiron which uses the Objective Caml type system to track most of the invariants of Chiron expressions statically. It includes a convenient foreign function interface (to facilitate building of biform theories out of existing libraries) and a sophisticated set of pretty-printers for rendering Chiron in ASCII, MathML, and LaTeX.

We also have an implementation of the **MathScheme Language**, which has a user-oriented, high-level syntax (unlike Chiron), influenced by our work on *high-level theories* [1]. The **MathScheme Library** is an experimental formalization of the theories of abstract algebra, basic data-structures, and structured type constructors. The library is organized by the *tiny theories method* in which knowledge is distributed over a network of theories that are built up one concept at a time. Much of the structure of the library resides in theory morphisms instead of in the theories themselves. We have an expander (to see what our theories correspond to in traditional notation), a type-checker, pretty-printing facilities similar to Chiron's, as well as an experimental translator to Chiron.

Eventually, the library will form a network of biform theories interconnected by theory morphisms. Some biform theories are *implementations* for developers while others are *interfaces* for users. Meta-programming techniques [2] will be used to generate efficient implementations from the MathScheme Language.

We are actively working on MathScheme. Current work is first focusing on further leveraging the structure already present in our library to automatically generate as much information as possible, rather than implementing all of this by hand, as is traditionally done.

References

1. Carette, J., Farmer, W.M.: High-level theories. In: Autexier, S., Campbell, J., Rubio, J., Sorge, V., Suzuki, M., Wiedijk, F. (eds.) AISC 2008, Calculemus 2008, and MKM 2008. LNCS (LNAI), vol. 5144, pp. 232–245. Springer, Heidelberg (2008)
2. Carette, J., Kiselyov, O.: Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code. Science of Computer Programming 76(5), 349–375 (2011)
3. Farmer, W.M.: Biform theories in Chiron. In: Kauers, M., Kerber, M., Miner, R.R., Windsteiger, W. (eds.) MKM/CALCULEMUS 2007. LNCS (LNAI), vol. 4573, pp. 66–79. Springer, Heidelberg (2007)
4. Farmer, W.M.: Chiron: A multi-paradigm logic. In: Matuszewski, R., Zalewska, A. (eds.) From Insight to Proof: Festschrift in Honour of Andrzej Trybulec. Studies in Logic, Grammar and Rhetoric, vol. 10(23), pp. 1–19. University of Białystok (2007)
5. Farmer, W.M.: Chiron: A set theory with types, undefinedness, quotation, and evaluation. SQRL Report No. 38, McMaster University (2007) (revised 2011)
6. MathScheme Web Site, http://www.cas.mcmaster.ca/research/mathscheme/
7. Ni, H.: Chiron: Mechanizing Mathematics in OCaml. Master's thesis, McMaster University (2009)
8. Objective Caml, http://www.caml.inria.fr/

Project Abstract: Logic Atlas and Integrator (LATIN)*

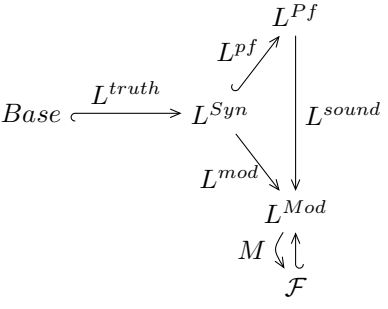
Mihai Codescu[1], Fulya Horozal[2], Michael Kohlhase[2], Till Mossakowski[1], and Florian Rabe[2]

[1] Safe and Secure Cognitive Systems, German Research Centre for Artificial Intelligence (DFKI), Bremen, Germany
[2] Computer Science, Jacobs University Bremen, Germany
http://latin.omdoc.org

LATIN aims at developing methods, techniques, and tools for interfacing logics and related formal systems. These systems are at the core of mathematics and computer science and are implemented in systems like (semi-)automated theorem provers, model checkers, computer algebra systems, constraint solvers, or concept classifiers. Unfortunately, these systems have differing domains of applications, foundational assumptions, and input languages, which makes them non-interoperable and difficult to compare and evaluate in practice.

The LATIN project develops a foundationally unconstrained framework for the representation of logics and translations between them [9,1]. The LATIN framework (i) subsumes existing proof theoretical frameworks such as LF and model theoretical frameworks such as institutions [3] and (ii) supplants them with a uniform knowledge representation language based on OMDoc. Special attention is paid to generality, modularity, scalability, extensibility, and interoperability.

Logics are represented as theories and translations as theory morphisms. Logic representations formalize the syntax, proof theory, and model theory of a logic within the LATIN framework. The representations of the model theory are parametric in the foundation of mathematics, which is represented as a theory itself; then individual models are represented as theory morphisms into the foundation. This can be represented in a diagram such as the one above, where the syntax of a logic L is represented as a theory L^{Syn}, which is then extended with the representation of proof rules to represent the proof theory as L^{pf}. Moreover, the model theory of the logic can be represented as a theory L^{Mod}, based on the representation of a foundation $\mathcal{F}$ which is included the model theory; the models are represented by the arrow M. The $Base$ theory represents the type of formulas and the notion



* The LATIN project is supported by the Deutsche Forschungsgemeinschaft (DFG) within grant KO 2428/9-1.

J.H. Davenport et al. (Eds.): Calculemus/MKM 2011, LNAI 6824, pp. 289–291, 2011.

of truth for them. Moreover, we can represent soundness proofs as a morphism L^{sound} from the proof theory to the model theory of L. Similarly, logic translations formalize the translations of syntax, proof theory, and model theory. This "logics-as-theories" approach makes system behaviors as well as their represented knowledge interoperable and thus comparable at multiple levels.

The LATIN framework has been implemented generically within the Heterogeneous Tool Set Hets [7] and instantiated with the logical frameworks LF, Isabelle, and Maude. Hets is a general institution-based framework for integration of formal methods and heterogeneous specification and proof management. While Hets implements a large number of logics and translations, their semantics and correctness had previously been determined only by model theoretic arguments. Within the LATIN project, Hets has been extended to support adding logics semi-automatically using a logic specification in one of the supported logical frameworks. This brings the advantage that the logics of Hets are represented fully formally and verified mechanically, and that new logics can be added dynamically.

To evaluate the developed framework and provide a service to the community, the project builds an atlas of logics used in automated reasoning, mathematics, and software engineering. The concrete logic representations span over 1000 theories and morphisms and can be found at the project web site, they include (i) *Type theory*, including a modular development of the lambda cube, Martin-Löf Type Theory, and Isabelle, (ii) *Logics*, including first-order, higher-order, modal, and description logics, (iii) *Set theory* including ZFC and the Mizar variant of Tarski-Grothendieck set theory. The atlas also includes a growing number of logic translations including, e.g., the relativization translations from modal, description, and sorted first-order logics to unsorted first-order logic, the interpretation of type theory in set theory, the negative translation from classical to intuitionistic logic, and the translation from first to higher-order logic. Elaborate case studies were documented in [4,5,10]. The LATIN atlas is extensible, and new logics can be added easily — including the reuse of already formalized logic features — and related to the existing logics via translations.

To make the logic atlas scalable, we base it on the knowledge representation language MMT [11]. MMT refines the markup language for structured theories that is part of OMDoc and provides a formal semantics for it. Moreover, MMT comes with a scalable infrastructure [6] centered around a flexible and foundation-independent API.

In the authoring work flow of LATIN, representations are written in Twelf [8] using our module system for it [12]. Twelf converts the content into OMDoc/MMT, which indexes and stores it in the SVN+XML database TNTBase [13]. In the application work flow, these OMDoc/MMT documents are imported into Hets. In the presentation work flow, the MMT web server uses XQueries to retrieve LATIN content and user-defined notations from TNTBase, which are used to render the content as JOBAD-enabled [2] XHTML+MathML. All stages of this pipeline are semantics-aware so that, for example, the web server can offer interactive dynamic services such as definition lookup or toggling the display of inferred types.

The browsable version of the atlas is available at the project web site. When browsing it, keep in mind that the logical framework, the formalizations in it, and the whole infrastructure processing them are ongoing work and thus subject to both constant improvement and temporary failures.

References

1. Codescu, M., Horozal, F., Kohlhase, M., Mossakowski, T., Rabe, F., Sojakova, K.: Towards Logical Frameworks in the Heterogeneous Tool Set Hets. In: Workshop on Abstract Development Techniques. LNCS. Springer, Heidelberg (to appear, 2011)
2. Gičeva, J., Lange, C., Rabe, F.: Integrating Web Services into Active Mathematical Documents. In: Carette, J., Dixon, L., Sacerdoti Coen, C., Watt, S. (eds.) MKM 2009, Held as Part of CICM 2009. LNCS, vol. 5625, pp. 279–293. Springer, Heidelberg (2009)
3. Goguen, J., Burstall, R.: Institutions: Abstract model theory for specification and programming. Journal of the Association for Computing Machinery 39(1), 95–146 (1992)
4. Horozal, F., Rabe, F.: Representing Model Theory in a Type-Theoretical Logical Framework. Theoretical Computer Science (to appear, 2011), http://kwarc.info/frabe/Research/HR_folsound_10.pdf
5. Iancu, M., Rabe, F.: Formalizing Foundations of Mathematics. Mathematical Structures in Computer Science (to appear, 2011), http://kwarc.info/frabe/Research/IR_foundations_10.pdf
6. Kohlhase, M., Rabe, F., Zholudev, V.: Towards MKM in the Large: Modular Representation and Scalable Software Architecture. In: Autexier, S., Calmet, J., Delahaye, D., Ion, P., Rideau, L., Rioboo, R., Sexton, A. (eds.) AISC 2010. LNCS, vol. 6167, pp. 370–384. Springer, Heidelberg (2010)
7. Mossakowski, T., Maeder, C., Lüttich, K.: The Heterogeneous Tool Set. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 519–522. Springer, Heidelberg (2007)
8. Pfenning, F., Schürmann, C.: System description: Twelf - a meta-logical framework for deductive systems. In: Ganzinger, H. (ed.) CADE 1999. LNCS (LNAI), vol. 1632, pp. 202–206. Springer, Heidelberg (1999)
9. Rabe, F.: A Logical Framework Combining Model and Proof Theory. Submitted to Mathematical Structures in Computer Science (2010), http://kwarc.info/frabe/Research/rabe_combining_09.pdf
10. Rabe, F.: Representing Isabelle in LF. In: Crary, K., Miculan, M. (eds.) Logical Frameworks and Meta-Languages: Theory and Practice. EPTCS, vol. 34, pp. 85–100 (2010)
11. Rabe, F., Kohlhase, M.: A Scalable Module System. Under review (2011), http://arxiv.org/abs/1105.0548
12. Rabe, F., Schürmann, C.: A Practical Module System for LF. In: Cheney, J., Felty, A. (eds.) Proceedings of the Workshop on Logical Frameworks: Meta-Theory and Practice (LFMTP), pp. 40–48. ACM Press, New York (2009)
13. Zholudev, V., Kohlhase, M.: TNTBase: a Versioned Storage for XML. In: Proceedings of Balisage: The Markup Conference 2009. Balisage Series on Markup Technologies, vol. 3. Mulberry Technologies, Inc., (2009)

The LaTeXML Daemon: Editable Math on the Collaborative Web

Deyan Ginev[1], Heinrich Stamerjohanns[1],
Bruce R. Miller[2], and Michael Kohlhase[1]

[1] Computer Science, Jacobs University Bremen
{first initial.last name}@jacobs-university.de
[2] National Institute of Standards and Technology
bruce.miller@nist.gov

Introduction. The language of the TEX/LATEX typesetting system has become all-pervasive in scientific publications and has proven its stability, convenience and expressivity in its three-decade history. With the advent of the Web 2.0 paradigm, it has also become the primary choice of various technical and scientific social platforms, most prominently online encyclopedias (e.g. PlanetMath [Pla]) and question-answer forums (e.g. MathOverflow [Mat]). On the other hand, the standardization of MathML and OpenMath and the adoption of the former in HTML5, have opened the floodgates for scientific content native to the browser.

LATEX on the Web. The efforts of bringing LATEX to the web are numerous and have varied in number and approach. Classical scenarios provide hooks to either LATEX itself or a LATEX daemon(e.g. [Uni]), incorporating formulas as PDF or PNG, while newer applications pursue a fully native output of XHTML+MathML (see [Sta+09] for an overview). The LaTeXML [Mil] system, and particularly its enhanced branch maintained for the arXMLiv [Sta+10] project, belongs to the second category. It is currently the only system with experimental Content MathML support and is also prepared for further linguistic and semantic analysis of the document contents, as it not only expands TEX macros, but also allows for the definition of custom bindings that preserve and enhance the semantic information encoded by the authors. Moreover, it takes the effort one step further, being able to already create XHTML+MathML+RDFa and remains easily extensible to other output formats, such as HTML5+RDFa.

A Daemon. In this abstract we present an extension, the LaTeXML daemon, which drastically widens the applications of the LaTeXML system, addressing the problems of *efficient*, *scalable* and *on-the-fly* processing. The daemon enhancements avoid the overhead of startup time and LATEX package initialization, achieving close to instant conversion times, and easily scale to multi-threaded setups. This provides a great boost to conversion times not only in the processing of large-scale corpora, but also when employed as a backend for web services using LATEX as a frontend language. Maturity has been exhibited by converting one and a half million abstracts from the ZentralblattMATH [Zbl] database, coupled with a stable performance in various installations of the Planetary [Koh+11] system.

J.H. Davenport et al. (Eds.): Calculemus/MKM 2011, LNAI 6824, pp. 292–294, 2011.

Next, we equip our system with the capabilities of operating as a web service independent of a file system and of recognizing resources via web-accessible URIs[1]. Furthermore, we add native support for user-embedded metadata, with an outlet for Semantic Web target representations, such as RDFa. When these features are employed in unity, our system acts as a capable conversion backend for web-based authoring systems, also scalable to exporting user-defined metadata for add-on semantic services. Prominently, the high conversion speeds allow for on-the-fly preview of the authored content and enable real-time collaborative setups, where multiple authors can write simultaneously – a setting explored in depth by services such as Etherpad [Inc] and Google Docs [Goo], but lacking native mathematical content.

Implementation. An emphasis has been placed on *flexibility*, *versatility* and *ease of use*, in both the setup and the deployment phases. A large range of customization options and a pair of intuitive server-client executables with detailed documentation, enable out-of-the-box use for a wide range of applications. The system implementation is based entirely on open web standards and has the full expressivity of the original TEX engine. Additionally, *correctness* and *robustness* are ensured respectively, via the powerful scoping system of LaTeXML and Perl's mastery in localizing both variables and processing flows. The daemon communication is based on sockets, allowing an easy coupling with both local and Internet services.

LaTeXML is Public Domain software, and the daemon remains consistently under that license. Currently the software is hosted on the arXMLiv branch of the LaTeXML repository [Gin], to be merged with the trunk of LaTeXML upon reaching a release candidate. Also, a demo page [Arx] has been set up, in order to showcase the features claimed.

Future Work. Our next steps would be to address reducing the memory and processing footprints in the short term, plus ensuring ironclad security when deployed on the web, in the mid-term. We are simultaneously developing a number of custom libraries that further build on the capabilities discussed, trying to fully realize the potential of the framework in introducing LATEX as a frontend language for the Collaborative Web. They range from supporting special input methods, e.g. Wikipedia style markup, to providing a stronger integration with the World Wide Web, e.g. by increasing the expressivity in writing metadata and supporting direct reuse of LATEX content available online.

References

[Arx] arXMLiv: Showcase Demo Page, `http://trac.kwarc.info/arXMLiv/wiki/Demo` (visited on 03/03/2011)

[Gin] Ginev, D.: LaTeXML: A LATEX to XML Converter, arXMLiv branch, `https://svn.mathweb.org/repos/LaTeXML/branches/arXMLiv` (visited on 03/03/2011)

[1] Note that in this respect LaTeXML exceeds the capability of TEX/LATEX which can only read files from the local file system.

[Goo] Google. Google Docs, http://docs.google.com (visited on 03/03/2011)
[Inc] AppJet Inc., Etherpad, http://www.etherpad.com (visited on 03/03/2011)
[Koh+11] Kohlhase, M., et al.: The Planetary System: Web 3.0 & Active Documents for STEM. accepted for publication at ICCS 2011 (Finalist at the Executable Papers Challenge) (2011), https://svn.mathweb.org/repos/planetary/doc/epc11/paper.pdf
[Mat] Google. Google Docs, http://docs.google.com (visited on 03/03/2011)
[Mil] Miller, B.: LaTeXML: A LATEX to XML Converter, http://dlmf.nist.gov/LaTeXML/ (visited on 03/03/2011)
[Pla] PlanetMath.org – Math for the people, by the people, http://planetmath.org (visited on 01/06/2011)
[Sta+09] Stamerjohanns, H., et al.: MathML-aware article conversion from LATEX, A comparison study. In: Sojka, P. (ed.) Towards Digital Mathematics Library, DML 2009 workshop, pp. 109–120. Masaryk University, Brno (2009), http://kwarc.info/kohlhase/submit/dml09.pdf
[Sta+10] Google. Google Docs, http://docs.google.com (visited on 03/03/2011)
[Uni] Open University. MathTran: A TEX to Image Converter Web Service (May 2010), http://www.mathtran.org (visited on 03/03/2011)
[Zbl] Kohlhase, M., et al.: The Planetary System: Web 3.0 & Active Documents for STEM. accepted for publication at ICCS 2011 (Finalist at the Executable Papers Challenge) (2011), https://svn.mathweb.org/repos/planetary/doc/epc11/paper.pdf

A System for Computing and Reasoning in Algebraic Topology*

Jónathan Heras, Vico Pascual, and Julio Rubio

Department of Mathematics and Computer Science of University of La Rioja
{jonathan.heras,vico.pascual,julio.rubio}@unirioja.es

Abstract. In this paper we present the *fKenzo* system, an *integral* assistant for teaching and research in (a subset of) Algebraic Topology. The *fKenzo* system provides a friendly graphical user interface which allows the user to interact with both the *Kenzo* and *GAP* Computer Algebra systems and, also, with the *ACL2* Theorem Prover by means of an intermediary layer based on XML technology.

System Description

Algebraic Topology is a mathematical subject which studies topological spaces using algebraic means, in particular through algebraic invariants (groups or rings, usually). This allows one to study interesting properties about topological spaces using statements about groups which are often easier to prove.

The *fKenzo* system [6] has been developed with the aim of being an *integral* assistant for research and teaching in (a subset of) Algebraic Topology. The "integral" adjective means that this assistant not only provides a graphical interface for using computational kernels, but also guides the user in his interaction with the system, and as far as possible, produces certificates about the correctness of the computations performed.

The *fKenzo* system provides a friendly front-end allowing the interoperability among different sources for computation and deduction by means of an intermediary layer based on XML technology.

The main computational kernel of our system is *Kenzo* [2], a Common Lisp program devoted to Symbolic Computation in Algebraic Topology which was developed by F. Sergeraert, which allows an *fKenzo* user to compute homology and homotopy groups of spaces. In addition, *GAP* [1], a Computer Algebra system well-known for its contributions in the area of Computational Group Theory, and its *HAP* package [3], an homological algebra library developed by G. Ellis, have been integrated in *fKenzo* allowing computations related to group homology. From the theorem proving side, *ACL2* [7], a first order logic theorem prover tool, is the core for verifying the correctness of statements.

In addition, we can say that the final aim of *fKenzo* has consisted not only in integrating several Computer Algebra systems and Theorem Prover tools, and

* Partially supported by Ministerio de Educación y Ciencia, project MTM2009-13842-C02-01.

J.H. Davenport et al. (Eds.): Calculemus/MKM 2011, LNAI 6824, pp. 295–297, 2011.

use them individually by means of a common GUI, but also in making them work in a coordinate and collaborative way to obtain new tools and results not reachable if we use individually each system.

As an example of this interoperability among systems, inspired by the work presented in [8], *Kenzo* and *GAP* have been combined in *fKenzo* to construct some spaces, namely Eilenberg MacLane spaces of type $K(G,1)$ where G is a cyclic group, which are instrumental in the computation of homotopy groups. The methodology presented in [8] to compute the homology groups of those Eilenberg MacLane spaces can be summed up as follows:

1. Load the necessary packages and files in GAP and Kenzo,
2. build the cyclic group G in GAP,
3. build a resolution of the cyclic group G using the HAP package,
4. export from GAP the resolution into a file using the OpenMath format,
5. import the resolution to Kenzo,
6. build the cyclic group G in Kenzo (thanks to a new Kenzo module developed in [8]),
7. assign the resolution to the corresponding cyclic group G in Kenzo,
8. build the space $K(G,1)$ where G is the cyclic group in Kenzo,
9. compute the homology groups of $K(G,1)$.

This approach has some drawbacks. First of all, the user must install several programs and packages: GAP, its HAP package, the OpenMath package for GAP [9], an extension for this OpenMath package developed in [8], the Kenzo system and the new module developed in [8]. In addition, of course, the user must know how to mix all the ingredients in order to obtain the desired result. Moreover, some of the steps could be performed automatically by a computer program; for instance, the importation/exportation of the resolution from GAP to Kenzo.

On the contrary, the procedure that the user must follow using *fKenzo* is:

1. Load the GAP *fKenzo* module,
2. build the cyclic group G,
3. build the space $K(G,1)$,
4. compute the homology groups of $K(G,1)$.

As can be seen, this is a much simpler approach than the one presented in [8] from the user point of view. To deal with the importation/exportation of the resolution from GAP to Kenzo, the SCSCP protocol [4] has been used.

Moreover, the reliability of such construction is increased by means of the *ACL2* Theorem Prover. Namely, ACL2 is invoked from *fKenzo* to generate a certificate of the correctness of the implementation of the cyclic group G which is used as input to construct the Eilenberg MacLane space $K(G,1)$. Therefore, we can claim that *Kenzo*, *GAP* and *ACL2* work together to provide a powerful and reliable tool thanks to the *fKenzo* system.

We urge the interested reader to consult [5] where he can find several demos, related papers and a complete system description of *fKenzo*.

References

1. GAP - Groups, Algorithms, Programming - System for Computational Discrete algebra, http://www.gap-system.org
2. Dousson, X., Rubio, J., Sergeraert, F., Siret, Y.: The Kenzo program. Institut Fourier, Grenoble (1998), http://www-fourier.ujf-grenoble.fr/~sergerar/Kenzo/
3. Ellis, G.: HAP package for GAP (2009), http://www.gap-system.org/Packages/hap.html
4. Freundt, S., Horn, P., Konovalov, A., Lindon, S., Roozemond, D.: Symbolic Computation Software Composability Protocol (SCSCP) specification, version 1.3 (2009), http://www.symbolic-computation.org/scscp
5. Heras, J.: The *fKenzo* program. University of La Rioja (2010), http://www.unirioja.es/cu/joheras/fKenzo/
6. Heras, J., Pascual, V., Rubio, J., Sergeraert, F.: *fKenzo*: A user interface for computations in Algebraic Topology. Journal of Symbolic Computation 46(6), 685–698 (2011)
7. Kaufmann, M., Moore, J.S.: ACL2, http://www.cs.utexas.edu/users/moore/acl2/
8. Romero, A., Ellis, G., Rubio, J.: Interoperating between computer algebra systems: computing homology of groups with Kenzo and GAP. In: Proceedings 34th International Symposium on Symbolic and Algebraic Computation (ISSAC 2009), pp. 303–310 (2009)
9. Solomon, A., Costantini, M.: GAP package OpenMath (2009), http://www.gap-system.org/Packages/openmath.html

Learning2Reason

Daniel Kühlwein, Josef Urban, Evgeni Tsivtsivadze,
Herman Geuvers, and Tom Heskes*

Institute for Computing and Information Sciences,
Radboud University Nijmegen, The Netherlands
firstname.lastname@science.ru.nl
http://www.fnds.cs.ru.nl/fndswiki/Research/Learning2Reason

Abstract. In recent years, large corpora of formally expressed knowledge have become available in the fields of formal mathematics, software verification, and real-world ontologies. The Learning2Reason project aims to develop novel machine learning methods for computer-assisted reasoning on such corpora. Our global research goals are to provide good methods for selecting relevant knowledge from large formal knowledge bases, and to combine them with automated reasoning methods.

Introduction

In recent years, large corpora of formally expressed knowledge have become available in the fields of formal mathematics, software verification, and real-world ontologies. Examples of such formal mathematical libraries are the Mizar Mathematical Library[1], the Constructive Coq Repository at Nijmegen[2], the Flyspeck project[3], the The Archive of Formal Proofs[4], and the Mathematical Components project[5]. Real-world ontologies include for example the Cyc Knowledge Base[6], the Suggested Upper Merged Ontology (SUMO)[7], and the YAGO semantic knowledge base[8]. To derive new knowledge from them, one uses computer-assisted and automated reasoning methods, which typically use one-problem-at-a-time approaches (e.g., resolution and tableaux proving). This *symbolic* (or *deductive*) approach typically does not consider the knowledge contained in previous proofs, and it suffers from a fast-growing search space.

This makes computer-assisted reasoning a suitable domain for complementary *heuristic* (or *inductive*) AI approaches, like machine learning, which rely on

* We acknowledge support from the Netherlands Organization for Scientific Research, in particular Learning2Reason and a Vici grant (639.023.604).

1 http://www.mizar.org
2 http://c-corn.cs.ru.nl/
3 http://code.google.com/p/flyspeck/
4 http://afp.sourceforge.net/
5 http://www.msr-inria.inria.fr/Projects/math-components
6 http://cyc.com/cyc/cycrandd/technology/whatiscyc_dir/whatsincyc
7 http://www.ontologyportal.org/
8 http://www.mpi-inf.mpg.de/yago-naga/yago/

J.H. Davenport et al. (Eds.): Calculemus/MKM 2011, LNAI 6824, pp. 298–300, 2011.

learning from the large amount of previous knowledge to heuristically control the search space, e.g., by estimating the usefulness of existing lemmas for proving a new result. We propose to use state-of-the-art machine learning techniques, especially kernel-based methods that are ideally suited to take into account the structure of formulas and proofs (given as graphs or trees).

Project Goals

The `Learning2Reason` project aims to propose a theoretical framework and to develop novel machine learning algorithms suitable for formal, computer-assisted, and automated reasoning in the presence of large amount of previous knowledge. The main advantage of our approach is its ability to *learn* and *predict* structural dependencies between theorems and axioms.

For ordinary users of formal systems, software developed in `Learning2Reason` will provide improved mathematical advice using *learned* dependencies mined from the large knowledge bases. For developers of automated reasoning methods, the `Learning2Reason` project will provide learning algorithms that help the process of proof discovery and verification.

Tasks

Our focus so far has been on the following tasks:

1. Developing suitable (structural, semantic) kernels for classification in the formal mathematical domain [3].
2. Formulating the learning task as an instance of different paradigms (binary or multi-class classification, (multi-output) ranking).
3. Developing multi-output rankers for the domain.
4. Defining and creating suitable representations of the mathematical data, and obtaining precise datasets and benchmarks for learning from mathematical libraries.
5. Defining proper machine learning evaluation metrics.
6. Plugging the newly developed machine learning tools into reasoning meta-systems like the Machine Learner for Automated Reasoning (MaLARea) [4].
7. Evaluation of the combined learning/deductive systems on standard large-theory benchmarks like the MPTP Challenge[9].

First Results

We conducted first experiments on a subsets of the Mizar mathematical library[10]. We compared a regression based kernel with two other ranking algorithms: SNoW [1] in naïve Bayes mode and APRILS, a ranking method which is

[9] `http://www.tptp.org/MPTPChallenge/`
[10] Available at `http://www.mizar.org`

based on Latent Semantic Analysis [2]. On the test datasets, our kernel algorithm outperforms both methods[11].

As an example, we present a comparison of our newly developed MOR algorithm with SNoW's naïve Bayes on the MPTP Challenge[12] problems. We experimentally plugged the MOR algorithm into the MaLARea system, and compare its speed and precision with MaLARea running with naïve bayes (SNoW) as a learning algorithm. Only eight MaLARea iterations are run, in order to remove the gradual effect of using many different specifications, which can with sufficient time equalize any advice algorithm with a random one. The comparison is given in figure 1.

Run	Time Limit	Axiom Limit	Solved Total MOR	Solved Total SNoW
1	1 sec	0	57	58
2	1 sec	256	79	74
3	1 sec	256	83	74
4	1 sec	4	90	84
5	1 sec	8	117	99
6	1 sec	16	135	119
7	1 sec	32	140	127
8	1 sec	64	141	129

Fig. 1. Comparison between MaLARea-MOR and MaLARea-SNoW

Even though the MOR algorithm started with one problem less solved, it converges faster and solved 141 problems after eight iterations. With the same number of iterations, MaLARea with SNoW's naïve Bayes solved only 129 problems. The number of problems solved by the system after the sixth fast one-second run using the MOR-based premise selection outperforms any other non-learning system run in 21 hours on the MPTP Challenge problems. Further information and results can be found on our website `http://www.fnds.cs.ru.nl/fndswiki/Research/Learning2Reason`.

References

1. Carlson, A., Cumby, C., Rizzolo, N., Rosen, J.: SNoW user manual (1999)
2. Deerwester, S., Dumais, S.T., Furnas, G.W., Landauer, T.K., Harshman, R.: Indexing by latent semantic analysis. Journal of the American Society for Information Science 41(6), 391–407 (1990)
3. Tsivtsivadze, E., Urban, J., Geuvers, H., Heskes, T.: Semantic Graph Kernels for Automated Reasoning. In: SIAM Conference on Data Mining (2011)
4. Urban, J., Sutcliffe, G., Pudlák, P.: Malarea SG1-machine learner for automated reasoning with semantic guidance. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 441–456. Springer, Heidelberg (2008)

[11] to be published, also see `http://www.cs.ru.nl/~kuehlwein/pics/mptp.pdf` and `http://www.cs.ru.nl/~kuehlwein/pics/530.pdf`

[12] `http://www.cs.miami.edu/~tptp/MPTPChallenge/`

A Formalization of the C99 Standard in HOL, Isabelle and Coq

Robbert Krebbers and Freek Wiedijk

Institute for Computing and Information Sciences,
Radboud University Nijmegen,
Heyendaalseweg 135, 6525 AJ Nijmegen, The Netherlands

Abstract. We recently started the Formalin project to create a formal version of the C99 standard for the C programming language. We are writing three matching formalizations for the interactive theorem provers HOL4, Isabelle/HOL and Coq, that all closely follow the existing C99 standard text. The project runs from 2011 to 2015, and involves a full time PhD student, a half time researcher and several scientific advisors.

The project differs from existing work in that our aim is to formalize the *full* C99 standard. This means that we treat the C preprocessor, the C standard library, floating point arithmetic, and 'dirty' C features like signal handling and volatile variables. Importantly, this means we also treat embedded C programs without explicit input/output.

The Formalin project [14], with website `http://ch2o.cs.ru.nl/`, runs from May 2011 to May 2015. The research team consists of the following people:

Robbert Krebbers	*PhD student*	
Freek Wiedijk	*project leader*	
Herman Geuvers	*promotor*	
James McKinna		
Erik Poll		
Michael Norrish	*HOL advisor*	NICTA, Australia
Andreas Lochbihler	*Isabelle advisor*	KIT, Germany
Jean-Christophe Filliâtre	*Coq advisor*	CNRS, France

The first five people are from the Radboud University in The Netherlands. The first two people are the developers, while the other six are advising.

The C programming language [8] is one of the most popular in the world. It is among the two currently most popular languages [9,13], and is a dominant language from the smallest microcontroller with only a few hundred bytes of RAM to the largest supercomputer that runs at petaflops speeds. C is especially used for embedded software, but it is also the native language of most modern operating systems due to its tight connection to Unix.

The current official description of the syntax and semantics of the C language – the C99 standard [5], issued by ANSI and ISO together – is written in English and does not use a mathematically precise formalism. This makes it inherently

J.H. Davenport et al. (Eds.): Calculemus/MKM 2011, LNAI 6824, pp. 301–303, 2011.

incomplete and ambiguous. Our project is to create a mathematically precise version of the C99 standard. We *formalize* the standard using interactive theorem provers (proof assistants) and develop the C99 formalization in three matching versions, for the interactive theorem provers HOL4 [7], Isabelle/HOL [11] and Coq [4]. These formalizations will all be derived from a common master formalization, which will either be written for one of the three systems or using a fourth system like for instance the Ott tool [12]. Ott is a tool designed for defining language definitions, which can generate formalizations for all three of our systems. The whole formalization suite will be published as open source, under a BSD-style license. For dissemination, we will also use MKM tools like the ones that currently are being developed in the MathWiki project [6].

The formalizations that we create closely follow the existing C99 standard text. Specifically we treat the C preprocessor, the C standard library, and features that in a formal treatment are often left out: unspecified and undefined behavior due to unknown evaluation order, casts between pointers and integers, alignment requirements, floating point arithmetic, non-local control flow (`goto` statements, `setjmp`/`longjmp` and signal handling) and volatile variables. Most importantly, to make our work relevant for verification of embedded software, we include the part of the standard referring to C programs that run in a 'free-standing environment' (Section 5.1.2.1 of [5]).

A formal version of the full C standard is an important artifact. When establishing a property of a C program, it is very attractive to be able to claim that it has been proved with respect to the *full* official standard. This kind of 'knowledge' about the C semantics in the current state of the art is mostly implicit in various tools.

A formalization of the C99 standard has three main applications:

- The C99 formalization makes the C99 standard utterly precise. This is useful for compiler writers, who will get the means to establish how the standard needs to be understood without having to deal with the ambiguities of the English language. Programmers writing C programs get the same benefit.
- There already are various projects to prove C compilers correct, like the Compcert project of Xavier Leroy [2]. These projects need a semantics of a version of C. These currently are subsets of full C, with names like Clight or C0. With a formal version of the C99 semantics, the correctness of the compiler becomes provable with respect to the full *official* standard.
- Currently people proving C programs correct with proof assistants use tools like VCC [3] and Frama-C [10] which generate *verification conditions* from C source annotated in the style of Hoare logic. These tools implicitly 'know' about the semantics of C, but this knowledge is not explicit. A more thorough approach is to have such a tool not just generate the verification conditions, but to also have it synthesize formal *proofs* about the properties of the program.

Our three formalizations each consist of two parts. The first part defines a space of all possible C semantics as a type `C_semantics`. ('Semanticses' is not correct English, but we mean the plural of semantics here.) Points in this space

correspond to various variants of C like the C99 standard, the upcoming C1X standard, and the behavior of specific C compilers on specific machines. The definition of this space will be as short as we can make it, to have it as clear as possible what our formalized C99 standard amounts to. This space also addresses the observation from page 70 of [1] that 'the C language does not exist'. The second part is a small step structured operational semantics of C. It corresponds to a *point* in the space of C semantics, which means that the second and main part of our formalizations will be a formal definition of an element

```
C99 : C_semantics
```

References

1. Bessey, A., et al.: A few billion lines of code later: using static analysis to find bugs in the real world. Communications of the ACM 53(2), 66–75 (2010)
2. Blazy, S., Leroy, X.: Mechanized semantics for the Clight subset of the C language. Journal of Automated Reasoning 43(3), 263–288 (2009)
3. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A Practical System for Verifying Concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009)
4. Coq Development Team. The Coq Proof Assistant Reference Manual (2010)
5. International Organization for Standardization. ISO/IEC 9899: Programming languages – C. ISO Working Group 14 (1999), Draft standard WG14/N1256, the combined C99 + TC1 + TC2 + TC3 (September 7, 2007)
6. Foundations Group of the ICIS. Research/MathWiki, http://www.fnds.cs.ru.nl/fndswiki/Research/MathWiki
7. Gordon, M., Melham, T. (eds.): Introduction to HOL. Cambridge University Press, Cambridge (1993)
8. Kernighan, B.W., Ritchie, D.M.: The C Programming Language, 2nd edn. Prentice Hall, Englewood Cliffs (1988)
9. LangPop.com. Programming Language Popularity, http://langpop.com/
10. Moy, Y., Marché, C.: Jessie Plugin Tutorial, Beryllium version. INRIA (2009)
11. Nipkow, T., Paulson, L., Wenzel, M. (eds.): Isabelle/HOL — A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002)
12. Sewell, P., et al.: Ott: Effective tool support for the working semanticist. Journal of Functional Programming 20(1), 70–122 (2010)
13. TIOBE Software. TIOBE Programming Community index, http://www.tiobe.com/content/paperinfo/tpci/
14. Wiedijk, F.: Formalizing the C99 standard in HOL, Isabelle and Coq (2010), http://www.cs.ru.nl/~freek/notes/ch2o.pdf

Krextor – An Extensible Framework for Contributing Content Math to the Web of Data

Christoph Lange

Computer Science, Jacobs University Bremen
ch.lange@jacobs-university.de

1 Problem Statement: No Math on the Web of Data Yet

An increasing amount of scientific knowledge is being contributed to the emerging Web of [Linked] Data, where it is made available in a machine-comprehensible way and interlinked with other related datasets. This already powers distributed query answering engines and intelligent semantic mashups enriching web publications – however, it still largely lacks mathematical functionality.[1] There are e-science datasets – with mathematical model descriptions opaque to machines. There are statistical datasets, e.g. from e-government – without explicit descriptions of how values have been derived. There are digital libraries and databases of scientific publications – with information about who cited your paper, but not who is building on your mathematical *ideas*.

2 The Krextor XML→RDF Extraction Library

In contrast to the document-oriented, often XML-based content markup of MKM, the graph-based RDF data model is most widely used for representing knowledge on the Web of Data. Therefore, in order to contribute mathematical knowledge to the Web of Data, we have developed the Krextor [3] library, and, on top of that, extraction modules that translate the structural outlines of OpenMath, OMDoc, and other content markup to RDF. Krextor is an XSLT library that aims at facilitating the repetitive task of implementing translations from several XML input languages. It does so by offering convenience templates and functions for frequently occurring patterns in XML→RDF translation, such as creating RDF resources for things represented by XML elements, generating ("minting") linked data compliant URIs for these resources, and translating XML text nodes or attribute to properties of these resources. Krextor allows for flexible integration into applications by supporting multiple output serializations of the RDF extracted, including callbacks to Java application code – whereas traditional hard-coded XSLT implementations would rather translate

[1] For a review of the state of the art of linked data, we refer to [2], and for further background about the potential of *mathematical* linked data to [6].

J.H. Davenport et al. (Eds.): Calculemus/MKM 2011, LNAI 6824, pp. 304–306, 2011.

from exactly one XML input language to exactly one RDF output serialization (e.g. RDF/XML). Besides OpenMath and OMDoc[2], we have developed extraction modules for special MKM applications, such as encoding semantic web ontologies in OMDoc, and external developers have adopted Krextor outside of MKM [3].

3 Publishing the OpenMath CDs as Linked Data

In contrast to previous work [5], the current focus of Krextor development is on expanding the coverage of the OpenMath 2 CD language (and proposed extensions beyond that), in order to prepare the publication of the official CDs at `openmath.org` as linked open data [7]. The official OpenMath CDs have a great potential for bootstrapping a mathematical Web of Data, as they are widely in use, e.g. in that they define the semantics of Content MathML 3 [1].

Krextor has been used with OpenMath CDs before, but specifically for maintaining the (then) experimental collection of "OpenMath/MathML 3 CDs" *inside* a closed semantic wiki [8], which pre-processed them in a special way. The current focus is on making most out of the official OpenMath CDs *as they are*, which means:[3] (i) Supporting the maintenance of links from concepts in the OpenMath CDs to semantically equivalent concepts in related datasets – such as the Digital Library of Mathematical Functions (DLMF [9]) or the PlanetMath encyclopedia [11]. As the reference encoding of the OpenMath CD model [12] does not currently have annotation facilities, this is done as standoff markup in separate RDF files next to the CDs.[4] An example is the definition of the sine function in terms of the exponential function ($\sin z = \frac{e^{iz}-e^{-iz}}{2i}$); the correspondence between its OpenMath and DLMF representations is expressed by the RDF triple `<http://dlmf.nist.gov/4.14.E1> owl:sameAs <http://www.openmath.org/cd/transc1#sin.prop0>`. (ii) A prerequisite for that: Giving stable identifiers to mathematical properties of symbols – even though the reference CD encoding does not provide such identifiers. This is important as, for example, the DLMF entries mainly correspond to OpenMath mathematical properties [7]. (iii) Utilizing existing XSLT code for translating OpenMath objects into Content MathML and other machine-comprehensible representations [10], so that interested applications can retrieve them right from the same dataset.

[2] cf. [4, chapter 3] for a detailed description of the target RDF vocabularies/ontologies that we have developed for capturing mathematical knowledge, or for mappings to existing RDF vocabularies that we reused, e.g. for metadata.

[3] See `http://trac.kwarc.info/krextor/wiki/OpenMathExtractionModule` for a technical documentation and examples.

[4] These links have to be maintained manually for now; automatically identifying such correspondences between would require advanced linguistic methods.

4 Coverage of the System Demo

The demo will focus on (i) the OpenMath CD extraction module, but also on (ii) Krextor's possibilities for implementing extraction modules for new MKM languages. Regarding (i), I will particularly explain how to create new links between the OpenMath CDs and external datasets, and how RDF- and/or OpenMath-aware client applications can utilize the OpenMath CD linked dataset. Regarding (ii), I am prepared for a "hacking session" with any visitors who are interested in extracting RDF from their XML-based MKM language, in order to contribute their mathematical knowledge collections to the Web of Data.

References

1. Mathematical Markup Language (MathML) Version 3.0. W3C Recommendation (2010), http://www.w3.org/TR/MathML3
2. Heath, T., Bizer, C.: Linked Data: Evolving the Web into a Global Data Space. Morgan & Claypool (2011), http://linkeddatabook.com
3. Krextor – The KWARC RDF Extractor, http://kwarc.info/projects/krextor/ (visited on 12/06/2010)
4. Lange, C.: Enabling Collaboration on Semiformal Mathematical Knowledge by Semantic Web Integration. submitted January 31, defended March 11. PhD thesis. Jacobs University Bremen (2011), https://svn.kwarc.info/repos/swim/doc/phd/phd.pdf (2011)
5. Lange, C.: Krextor – An Extensible XML⟶RDF Extraction Framework. In: Scripting and Development for the Semantic Web (SFSW). CEUR Workshop Proc., vol. 449 (2009), http://CEUR-WS.org/Vol-449/
6. Lange, C.: Ontologies and Languages for Representing Mathematical Knowledge on the Semantic Web. Semantic Web Journal (2011), http://www.semantic-web-journal.net/content/new-submission-ontologies-and-languages-representingmathematical-knowledge-semantic-web (accepted)
7. Lange, C.: Towards OpenMath Content Dictionaries as Linked Data. In: 23rd OpenMath Workshop, arXiv:1006.4057v1 [cs.DL] (2010)
8. Lange, C., González Palomo, A.: Easily Editing and Browsing Complex OpenMath Markup with SWiM. In: Mathematical User Interfaces Workshop (2008), http://www.activemath.org/workshops/MathUI/08/proceedings/LangeGonzales-OMEdit.html
9. National Institute of Standards and Technology, ed. Digital Library of Mathematical Functions (May 7, 2010), http://dlmf.nist.gov
10. OpenMath Society, http://www.openmath.org/standard/omxsl/ (visited on 08/09/2010)
11. PlanetMath.org – Math for the people, by the people, http://planetmath.org (visited on 01/06/2011)
12. The Open Math Standard, Version 2.0. Tech. rep. The OpenMath Society (2004), http://www.openmath.org/standard/om20

System Description: EgoMath2 As a Tool for Mathematical Searching on Wikipedia.org*

Jozef Mišutka[1] and Leo Galamboš[2]

[1] Department of Software Engineering, Charles University in Prague
misutka@ksi.mff.cuni.cz
[2] Cythres group, Czech Technical University in Prague
galamleo@fd.cvut.cz

1 Introduction

EgoMath is a full text search engine focused on digital mathematical content with little semantic information available. Recently, we have decided that another step towards making mathematics in digital form more accessible was to enable mathematical searching in one of the world's largest digital libraries - Wikipedia. The library is an excellent candidate for our mathematical search engine because the mathematical notation is represented by TEX fragments which do not contain semantic information.

The key issue in mathematical searching is the retrieval of mathematically equal formulae. We regard this issue as a similarity search problem where the similarity function is strongly dependent on the mathematical model. EgoMath2 and its predecessor use the same idea but different implementation for the similarity function, indexing and searching. The idea is to use content-based annotation - different textual representations of one formula which are mathematically similar in our definition - for allowing similarity search. The similarity of mathematical formulae in EgoMath2 is based on the mathematical equality in a predefined mathematical model and preferring operations to operators in their formula parse trees. The later characteristic is used in generalising the formula representation e.g. formula $a + 7$ is generalised to $id + 7$. Currently used model consists of several rules e.g. commutative property of addition. One possible definition of similarity can be found in [1]. The similarity search is hidden in multiple queries which can be performed while searching for one formula in the space of equal and similar representations in the index.

A textual representation of a mathematical formula is a TEX-like flattened symbol representation by words e.g. a^2 is represented by three words: a, ^, 2, internally represented in postfix notation to avoid parenthesis issues. There are two important algorithms used during the content-based annotation. The *augmentation* algorithm exploits the biggest advantage of full text search engines - fast searching in a huge set of words. Consequently, for each formula the algorithm produces several representations consisting of ordered words which are

* This work was supported by the grant SVV-2011-263312.

J.H. Davenport et al. (Eds.): Calculemus/MKM 2011, LNAI 6824, pp. 307–309, 2011.

indexed like normal text. Each textual representation of a formula is equal to or less similar than the previous one.

The *ordering* algorithm converts each representation to a canonical one. The ordering algorithm guarantees that two mathematically equal formulae with the same but permuted operands have the same unique canonical representation. Two similar (but not equal) formulae have a similar textual representation.

Semantically rich mathematical formulae cannot be represented by a full text search engine without losing the semantic information in general. The augmentation and ordering try to minimise this disadvantage.

2 New Features and Architecture Changes in EgoMath2

EgoMath2 is based on the newest version of the Java full text search engine Egothor (http://www.egothor.org). Strongly decoupled architecture of the mathematical extension and the full text indexer made the update straightforward. Learning from our experience with the first version, the augmentation process in EgoMath2 was made easily extensible and configurable using XML configuration files. Both the algorithms which are applied during the augmentation and the ambiguous symbol meaning can be configured to take advantage of additional knowledge about the underlying document set. Architectural changes were made allowing for ranking the query results.

The graphical user interface(UI) had been completely rewritten [2], thus the mathematical support had to be implemented from scratch. The connection between indexer and the UI was simplified, a new text element for mathematical input was added, snippets showing matched formula representations were introduced and debugging capabilities were also improved. The new UI has administration features which can be useful in online mathematical systems. EgoMath2 supports roles with different privileges offering different search indexes. The indexer web administration has proven itself useful for quick document set inspection. The performance of the mathematical indexer was improved (EgoMath2 is 3x faster in indexing Wikipedia dataset than the previous version) by caching formulae string representations and by small optimisations and cleanup of code.

3 Adjusting Wikipedia for EgoMath2 and vice versa

Preparing Wikipedia for indexing by EgoMath2 means to download articles from Wikipedia.org, sort out non-mathematical articles, convert mathematical notation into supported format and create HTML pages which are fed to EgoMath2. A dump of English articles from January 2011 (30GB) was downloaded from the official website [3]. One by one, all types of articles were extracted. The mathematical articles were identified by looking for the string "<math>". 28,376 mathematical articles (425MB dump) have been found with more than 240,000 mathematical elements from more than 10 million articles. We tried to convert each element to semantically richer MathML using latex2mathml web

service developed by the KWARC group [4] to improve the semantic information. EgoMath2 then uses both the LaTeX and the MathML format of the formula if available. More than 300 new symbols (e.g. Invisible Separator U+2063) have been added into our XML symbol configuration to improve the semantic quality of the extracted mathematical formula. Several modifications had to be made because of incorrect conversions of complex formulae. The error checking had to be relaxed. The parser heuristics had to be improved because TeX fragments misused symbols and operators e.g. f^{'} denotes derivation. A maximum depth limit was introduced into one of the algorithms which computed canonical distributivity because independent tables and other structures were put into one TeX fragment and the algorithm complexity grew rapidly with the number of operators.

4 Conclusion and Availability

There are many digital scientific repositories with little semantic information available. We think that focusing on these repositories is very important because they will still prevail in the near future. We showed that mathematical searching in one of the world's most important one is feasible at least from the technical point of view. The focus was mainly on recall so the next step is to focus on precision and preferring more similar result. This means to start using the built-in ranking algorithm and gathering feedback from the users. The online version with additional description can be found at [5]. Administration credentials, sources to Wikipedia converters, dumps and document set are available upon request.

References

1. Mišutka, J., Galamboš, L.: Extending Full Text Search Engine for Mathematical Content. In: Towards Digital Mathematics Library, Birmingham, UK, pp. 55–67 (2008)
2. Tamáš, M.: WSE (2010), http://www.projects.eblend.net/web-search-engine/
3. Wikipedia: Database download, http://www.en.wikipedia.org/wiki/Wikipedia:Database_download
4. latex2mathml converter service, http://www.tex2xml.kwarc.info
5. EgoMath2 mathematical search engine, http://www.egomath.cythres.cz:8080/egomath/

Author Index